topologic Documentation

Release 0.1.9.dev20221013213052

Microsoft

Oct 13, 2022

CONTENTS:

CHAPTER

ONE

TOPOLOGIC LIBRARY DOCUMENTATION

1.1 topologic package

topologic.**connected_components_generator**(*graph: networkx.classes.graph.Graph*) → Generator[networkx.classes.graph.Graph, None, None]

Returns a Generator that will provide each component as a networkx.Graph copy

Parameters **graph** (*networkx.Graph*) – The networkx graph object to create a connected component generator from

Returns A Generator that returns a copy of the subgraph corresponding to a connected component of *graph*

Return type Generator[networkx.Graph]

exception topologic.**DialectException**(*message*)

Bases: BaseException

exception topologic.**InvalidGraphError**(*message*)

Bases: BaseException

topologic.**largest_connected_component**(*graph: networkx.classes.graph.Graph*, *weakly: bool = True*) → networkx.classes.graph.Graph

Returns the largest connected component of the graph.

Parameters

- **graph** (*networkx.Graph*) – The networkx graph object to select the largest connected component from. Can be either directed or undirected.
- **weakly** (*bool*) – Whether to find weakly connected components or strongly connected components for directed graphs.

Returns A copy of the largest connected component as an nx.Graph object

Return type networkx.Graph

topologic.**number_connected_components**(*graph: networkx.classes.graph.Graph*) → int

Returns the number of connected components in the Graph.

This function calls the appropriate newtorkx connected components function depending on whether it is Undirected or Directed.

Parameters **graph** (*networkx.Graph*) – The networkx graph object to determine the number of connected components for

Returns number of connected components (and in the case of a directed graph, strongly connected)

Return type int

class topologic.**PartitionedGraph**
Bases: tuple

A PartitionedGraph combines a networkx graph object with a global community partitioning for that graph.

property community_partitions
Alias for field number 1

property graph
Alias for field number 0

topologic.**diagonal_augmentation**(*graph: Union[networkx.classes.graph.Graph, networkx.classes.digraph.DiGraph], weight_column: str = 'weight'*) → networkx.classes.graph.Graph
Replaces the diagonal of adjacency matrix of the graph with the weighted degree / number of vertices in graph. For directed graphs, the weighted in and out degree is averaged.

Modifies the provided graph in place as well as returning it.

Param The networkx graph which will get a replaced diagonal

Parameters **weight_column** (*str*) – The weight column of the edge

Returns The networkx Graph or DiGraph object that was modified in place.

Return type Union[nx.Graph, nx.DiGraph]

exception topologic.**UnweightedGraphError**(*message*)
Bases: BaseException

1.1.1 Subpackages

topologic.embedding package

topologic.embedding.**adjacency_embedding**(*graph: networkx.classes.graph.Graph, maximum_dimensions: int = 100, elbow_cut: Optional[int] = 1, weight_column: str = 'weight', svd_seed: Optional[int] = None, num_iterations: int = 5, power_iteration_normalizer: str = 'QR', num_oversamples: int = 10*) → topologic.embedding.embedding_container.EmbeddingContainer
Generates a spectral embedding based upon the adjacency matrix of the graph.

See also: https://csustan.csustan.edu/~tom/Clustering/GraphLaplacian-tutorial.pdf

Parameters

- **graph** (*networkx.Graph*) – graph_augmented_sparse_matrix networkx Graph object containing no more than one connected component. Note that if the graph is a directed graph, the resulting dimensionality of the embedding will be twice that of an undirected graph
- **maximum_dimensions** (*int*) – Maximum dimensions of embeddings that will be returned - defaults to 100. Actual dimensions of resulting embeddings should be significantly smaller, but will never be over this value.
- **elbow_cut** (*Optional[int]*) – scree plot elbow detection will detect (usually) many elbows. This value specifies which elbow to use prior to filtering out extraneous dimensions. If None, then an embedding of size *maximum_dimensions* will be returned.

- **weight_column** (*str*) – The weight column to use in the Graph.
- **svd_seed** (*Optional[int]*) – If not provided, uses a random number every time, making consistent results difficult Set this to a random int if you want consistency between executions over the same graph.
- **num_iterations** (*int*) – The number of iterations to be used in the svd solver.
- **num_oversamples** (*int*) – Additional number of random vectors to sample the range of M so as to ensure proper conditioning. The total number of random vectors used to find the range of M is n_components + n_oversamples. Smaller number can improve speed but can negatively impact the quality of approximation of singular vectors and singular values.
- **power_iteration_normalizer** (*Optional[str]*) – Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when *n_iter* is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if *num_iterations* <= 2 and switches to LU otherwise.

  Options: 'auto' (default), 'QR', 'LU', 'none'

Returns EmbeddingContainer containing a matrix, which itself contains the embedding for each node. the tuple also contains a vector containing the corresponding vertex labels for each row in the matrix. the matrix and vector are positionally correlated.

Return type *EmbeddingContainer*

class topologic.embedding.**EmbeddingContainer**(*embedding*, *vertex_labels*)

Bases: tuple

property embedding
Alias for field number 0

to_dictionary()

property vertex_labels
Alias for field number 1

class topologic.embedding.**EmbeddingMethod**

Bases: enum.Enum

An enum to represent which embedding method to use when generating an Omnibus embedding

ADJACENCY_SPECTRAL_EMBEDDING = 0

LAPLACIAN_SPECTRAL_EMBEDDING = 1

topologic.embedding.**find_elbows**(*iterable_to_search: Union[list, numpy.array]*, *num_elbows: int = 1*, *threshold: int = 0*) → numpy.array

An implementation of profile likelihood as outlined in Zhu and Ghodsi References, Zhu, Mu and Ghodsi, Ali (2006), Automatic dimensionality selection from the scree plot via the use of profile likelihood, Computational Statistics & Data Analysis, Volume 51 Issue 2, pp 918-930, November, 2006

Examples

```
>>> input_data = [2, 3, 4, 5, 6, 7, 8, 9]
>>> result: np.array = find_elbows(input_data, num_elbows=1,␣
↪threshold=0)
>>> result.size
1
```

Parameters

- **iterable_to_search** – An ordered or unordered list of values that will be used to find the elbows.
- **num_elbows** – The number of elbows to return
- **threshold** – Smallest value to consider. Nonzero thresholds will affect elbow selection.

Returns A numpy array containing elbows

topologic.embedding.**generate_omnibus_matrix**(*matrices: List[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*) → numpy.ndarray

Generate the omnibus matrix from a list of adjacency or laplacian matrices as described by 'A central limit theorem for an omnibus embedding of random dot product graphs.'

Given an iterable of matrices a, b, … n then the omnibus matrix is defined as:

```
[[           a, .5 * (a + b), ..., .5 * (a + n)],
 [.5 * (b + a),           b, ..., .5 * (b + n)],
 [         ...,         ..., ...,         ...],
 [.5 * (n + a),  .5 * (n + b, ...,           n]
]
```

The current iteration of this function operates in O(n) but a further optimization could take it to O(.5 * n)

See also: The original paper - https://arxiv.org/abs/1705.09355

Parameters **matrices** (*List[Union[numpy.ndarray, scipy.sparse.csr_matrix]]*) – The list of matrices to generate the Omnibus matrix

Returns An Omnibus matrix

topologic.embedding.**laplacian_embedding**(*graph: networkx.classes.graph.Graph*, *maximum_dimensions: int = 100*, *elbow_cut: Optional[int] = 1*, *weight_column: str = 'weight'*, *svd_seed: Optional[int] = None*, *num_iterations: int = 5*, *power_iteration_normalizer: str = 'QR'*, *num_oversamples: int = 10*) → topologic.embedding.embedding_container.EmbeddingContainer

Generates a spectral embedding based upon the Laplacian matrix of the graph.

See also: https://csustan.csustan.edu/~tom/Clustering/GraphLaplacian-tutorial.pdf

Parameters

- **graph** (*networkx.Graph*) – A networkx Graph object containing no more than one connected component. Note that if the graph is a directed graph, the resulting dimensionality of the embedding will be twice that of an undirected graph
- **maximum_dimensions** (*int*) – Maximum dimensions of embeddings that will be returned - defaults to 100. Actual dimensions of resulting embeddings should be significantly smaller, but will never be over this value.
- **elbow_cut** (*Optional[int]*) – scree plot elbow detection will detect (usually) many elbows. This value specifies which elbow to use prior to filtering out extraneous dimensions. If None, then an embedding of size *maximum_dimensions* will be returned.
- **weight_column** (*str*) – The weight column to use in the Graph.
- **svd_seed** (*Optional[int]*) – If not provided, uses a random number every time, making consistent results difficult Set this to a random int if you want consistency between executions over the same graph.

- **num_iterations** (*int*) – The number of iterations to be used in the svd solver.
- **num_oversamples** (*int*) – Additional number of random vectors to sample the range of M so as to ensure proper conditioning. The total number of random vectors used to find the range of M is n_components + n_oversamples. Smaller number can improve speed but can negatively impact the quality of approximation of singular vectors and singular values.
- **power_iteration_normalizer** (*Optional[str]*) – Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when *n_iter* is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if *num_iterations* <= 2 and switches to LU otherwise.

  Options: 'auto' (default), 'QR', 'LU', 'none'

Returns EmbeddingContainer containing a matrix, which itself contains the embedding for each node. the tuple also contains a vector containing the corresponding vertex labels for each row in the matrix. the matrix and vector are positionally correlated.

Return type *EmbeddingContainer*

topologic.embedding.**node2vec_embedding**(*graph: networkx.classes.graph.Graph*, *num_walks: int = 10*, *walk_length: int = 80*, *return_hyperparameter: int = 1*, *inout_hyperparameter: int = 1*, *dimensions: int = 128*, *window_size: int = 10*, *workers: int = 8*, *iterations: int = 1*, *interpolate_walk_lengths_by_node_degree: bool = True*) → topologic.embedding.embedding_container.EmbeddingContainer

Generates a node2vec embedding from a given graph. Will follow the word2vec algorithm to create the embedding.

Parameters

- **graph** (*networkx.Graph*) – A networkx graph. If the graph is unweighted, the weight of each edge will default to 1
- **num_walks** (*int*) – Number of walks per source. Default is 10.
- **walk_length** (*int*) – Length of walk per source. Default is 80.
- **return_hyperparameter** (*int*) – Return hyperparameter (p). Default is 1.
- **inout_hyperparameter** (*int*) – Inout hyperparameter (q). Default is 1.
- **dimensions** (*int*) – Dimensionality of the word vectors. Default is 128.
- **window_size** (*int*) – Maximum distance between the current and predicted word within a sentence. Default is 10.
- **workers** (*int*) – Use these many worker threads to train the model. Default is 8.
- **iterations** (*int*) – Number of epochs in stochastic gradient descent (SGD)
- **interpolate_walk_lengths_by_node_degree** (*bool*) – Use a dynamic walk length that corresponds to each nodes degree. If the node is in the bottom 20 percentile, default to a walk length of 1. If it is in the top 10 percentile, use walk_length. If it is in the 20-80 percentiles, linearly interpolate between 1 and walk_length.

  This will reduce lower degree nodes from biasing your resulting embedding. If a low degree node has the same number of walks as a high degree node (which it will if this setting is not on), then the lower degree nodes will take a smaller breadth of random walks when

compared to the high degree nodes. This will result in your lower degree walks dominating your higher degree nodes.

Returns tuple containing a matrix, which itself contains the embedding for each node. the tuple also contains a vector containing the corresponding vertex labels for each row in the matrix. the matrix and vector are positionally correlated.

Return type *EmbeddingContainer*

topologic.embedding.**omnibus_embedding**(*graphs: List[networkx.classes.graph.Graph], maximum_dimensions: int = 100, elbow_cut: Optional[int] = 1, embedding_method: topologic.embedding.embedding_methods.EmbeddingMethod = <EmbeddingMethod.LAPLACIAN_SPECTRAL_EMBEDDING: 1>, svd_seed: Optional[int] = None, num_iterations: int = 5, power_iteration_normalizer: str = 'QR', num_oversamples: int = 10*) → List[Tuple[topologic.embedding.embedding_container.EmbeddingContainer, topologic.embedding.embedding_container.EmbeddingContainer]]

Generates a pairwise omnibus embedding for each pair of graphs in a list of graphs. If given graphs A, B, and C, the embeddings will be computed for A,B and B,C.

There should be exactly the same number of nodes in each graph with exactly the same labels. The list of graphs should represent a time series and should be in an order such that time is continuous through the list of graphs.

If the labels differ between each pair of graphs, then those nodes will only be found in the resulting embedding if they exist in the largest connected component of the union of all edges across all graphs in the time series.

Parameters

- **graphs** (*List[networkx.Graph]*) – A list of graphs that will be used to generate the omnibus embedding. Each graph should have exactly the same vertices as each of the other graphs. The order of the graphs in the list matter. The first graph will be at time 0 and each following graph will increment time by 1.
- **maximum_dimensions** (*int*) – Maximum dimensions of embeddings that will be returned - defaults to 100. Actual dimensions of resulting embeddings should be significantly smaller, but will never be over this value.
- **elbow_cut** (*int*) – scree plot elbow detection will detect (usually) many elbows. This value specifies which elbow to use prior to filtering out extraneous dimensions.
- **embedding_method** (topologic.embedding.EmbeddingMethod) – The embedding technique used to generate the Omnibus embedding.
- **svd_seed** (*Optional[int]*) – If not provided, uses a random number every time, making consistent results difficult Set this to a random int if you want consistency between executions over the same graph.
- **num_iterations** (*int*) – The number of iterations to be used in the svd solver.
- **num_oversamples** (*int*) – Additional number of random vectors to sample the range of M so as to ensure proper conditioning. The total number of random vectors used to find the range of M is n_components + n_oversamples. Smaller number can improve speed but can negatively impact the quality of approximation of singular vectors and singular values.
- **power_iteration_normalizer** (*Optional[str]*) – Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when *n_iter* is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if *num_iterations* <= 2 and switches to LU otherwise.

Options: 'auto' (default), 'QR', 'LU', 'none'

Returns A List of EmbeddingContainers each containing a matrix, which itself contains the embedding for each node. the tuple also contains a vector containing the corresponding vertex labels for each row in the matrix. the matrix and vector are positionally correlated.

Return type List[(*EmbeddingContainer*, *EmbeddingContainer*)]

class topologic.embedding.**OutOfSampleEmbeddingContainer**(*embedding*, *vertex_labels*, *vertex_labels_failing_inference*, *starting_index_of_oos_embedding*, *u*, *sigma*)

Bases: tuple

property embedding
Alias for field number 0

property sigma
Alias for field number 5

property starting_index_of_oos_embedding
Alias for field number 3

to_dictionary()

property u
Alias for field number 4

property vertex_labels
Alias for field number 1

property vertex_labels_failing_inference
Alias for field number 2

topologic.embedding.**pca**(*embedding: numpy.ndarray, num_components: Union[str, int], whiten: bool = False, svd_solver: str = 'auto', tolerance: float = 0.0, iterated_power: Union[int, str] = 'auto', random_state: Union[int, numpy.random.mtrand.RandomState, None] = None*) → numpy.ndarray

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

Parameters

- **embedding** (*numpy.ndarray*) – The embedding in which PCA will be applied
- **num_components** (*Union[str, int]*) – If `num_components == 'mle'` and `svd_solver == 'full'`, Minka's MLE is used to guess the dimension. Use of `num_components == 'mle'` will interpret `svd_solver == 'auto'` as `svd_solver == 'full'`.

  If `0 < num_components < 1` and `svd_solver == 'full'`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by num_components.

  If `svd_solver == 'arpack'`, the number of components must be strictly less than the minimum of number of features and the number of samples.

- **whiten** (*bool*) – When True (False by default) the *components_* vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

  Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

- **svd_solver** (*str*) –

  auto : the solver is selected by a default policy based on *X.shape* and *num_components*: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

  full : run exact full SVD calling the standard LAPACK solver via *scipy.linalg.svd* and select the components by postprocessing

  arpack : run SVD truncated to num_components calling ARPACK solver via *scipy.sparse.linalg.svds*. It requires strictly 0 < num_components < min(X.shape)

  randomized : run randomized SVD by the method of Halko et al.

- **tolerance** (*float*) – Tolerance for singular values computed by svd_solver == 'arpack'. A float value >=0 with default 0

- **iterated_power** (*Union[int, str]*) – Number of iterations for the power method computed by svd_solver == 'randomized'.

- **random_state** (*Optional[int]*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `svd_solver` == 'arpack' or 'randomized'.

Returns A np.ndarray of principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by variance`

Return type numpy.ndarray

topologic.embedding.**sample_graph_by_edge_weight**(*graph*, *weight_column='weight'*, *weight_cutoff=None*, *percentage=0.1*, *nodelist=None*)

topologic.embedding.**sample_graph_by_vertex_degree**(*graph*, *degree_cutoff=None*, *percentage=0.1*, *nodelist=None*)

class topologic.embedding.**SampleMethod**

Bases: enum.Enum

An enumeration.

EDGE_WEIGHT = 1

VERTEX_DEGREE = 0

topologic.embedding.**tsne**(*embedding: numpy.ndarray*, *num_components: int = 2*, *perplexity: float = 30.0*, *early_exaggeration: float = 12.0*, *learning_rate: float = 200.0*, *num_iterations: int = 1000*, *num_iterations_without_progress: int = 300*, *min_grad_norm: float = 1e-07*, *metric: str = 'euclidean'*, *init: str = 'random'*, *verbose: int = 1*, *random_state: Union[int, numpy.random.mtrand.RandomState, None] = None*, *method: str = 'barnes_hut'*, *angle: float = 0.5*) → numpy.ndarray

t-distributed Stochastic Neighbor Embedding.

t-SNE is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high. This will suppress some noise and speed up the computation of pairwise distances between samples.

Parameters

- **embedding** (*numpy.ndarray*) – The embedding in which PCA will be applied
- **num_components** (*int*) – Dimension of the embedded space. Default 2
- **perplexity** (*float*) – The perplexity is related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50. The choice is not extremely critical since t-SNE is quite insensitive to this parameter. Default 30.0
- **early_exaggeration** (*float*) – Controls how tight natural clusters in the original space are in the embedded space and how much space will be between them. For larger values, the space between natural clusters will be larger in the embedded space. Again, the choice of this parameter is not very critical. If the cost function increases during initial optimization, the early exaggeration factor or the learning rate might be too high. Default 12.0
- **learning_rate** (*float*) – The learning rate for t-SNE is usually in the range [10.0, 1000.0]. If the learning rate is too high, the data may look like a 'ball' with any point approximately equidistant from its nearest neighbours. If the learning rate is too low, most points may look compressed in a dense cloud with few outliers. If the cost function gets stuck in a bad local minimum increasing the learning rate may help. Default 200.0
- **num_iterations** (*int*) – Maximum number of iterations for the optimization. Should be at least 250. Default 1000
- **num_iterations_without_progress** (*int*) – Maximum number of iterations without progress before we abort the optimization, used after 250 initial iterations with early exaggeration. Note that progress is only checked every 50 iterations so this value is rounded to the next multiple of 50. Default 300
- **min_grad_norm** (*float*) – If the gradient norm is below this threshold, the optimization will be stopped. Default 1e-7
- **metric** (*Union[str, Callable]*) – The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is "precomputed", X is assumed to be a distance matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them. The default is "euclidean" which is interpreted as squared euclidean distance. Default 'euclidean'
- **init** (*Union[string, numpy.ndarray]*) – Initialization of embedding. Possible options are 'random', 'pca', and a numpy array of shape (n_samples, num_components). PCA initialization cannot be used with precomputed distances and is usually more globally stable than random initialization. Default 'random'
- **verbose** (*int*) – Verbosity level. Default 1

- **random_state** (*Optional[Union[int, numpy.random.RandomState]]*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Note that different initializations might result in different local minima of the cost function.
- **method** (*str*) – By default the gradient calculation algorithm uses Barnes-Hut approximation running in O(NlogN) time. method='exact' will run on the slower, but exact, algorithm in O(N^2) time. The exact algorithm should be used when nearest-neighbor errors need to be better than 3%. However, the exact method cannot scale to millions of examples. Default 'barnes_hut'
- **angle** (*float*) – Only used if method='barnes_hut' This is the trade-off between speed and accuracy for Barnpcaes-Hut T-SNE. 'angle' is the angular size (referred to as theta in [3]) of a distant node as measured from a point. If this size is below 'angle' then it is used as a summary node of all points contained within it. This method is not very sensitive to changes in this parameter in the range of 0.2 - 0.8. Angle less than 0.2 has quickly increasing computation time and angle greater 0.8 has quickly increasing error. Default 0.5

Returns A np.ndarray of principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by variance`

Return type numpy.ndarray

Subpackages

topologic.embedding.clustering package

topologic.embedding.clustering.**dbscan**(*embedding: numpy.ndarray*, *eps: float = 0.5*, *min_samples: int = 5*, *metric: str = 'minkowski'*, *metric_params: dict = None*, *algorithm: str = 'auto'*, *leaf_size: int = 30*, *p: float = 2*, *sample_weight: array.array = None*, *n_jobs: int = None*) → numpy.ndarray

Perform DBSCAN clustering from vector array or distance matrix.

Parameters

- **embedding** (*numpy.ndarray*) – An n x d array of vectors representing n labels in a d dimensional space
- **eps** (*Optional[float]*) – The maximum distance between two samples for them to be considered as in the same neighborhood.
- **min_samples** (*Optional[int]*) – The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.
- **metric** (*Union[str, Callable[[float, float], float]]*) – The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by `sklearn.metrics.pairwise_distances()` for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

  If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them.

- **metric_params** (*Optional[dict]*) – Additional keyword arguments for the metric function.
- **algorithm** (*Optional[str]*) – The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. Potential values: {'auto', 'ball_tree', 'kd_tree', 'brute'}, optional
- **leaf_size** (*Optional[int]*) – Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Default 30
- **p** (*Optional[float]*) – The power of the Minkowski metric to be used to calculate distance between points. Default 2.0
- **sample_weight** (*Optional[Array[int]]*) – Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.
- **n_jobs** (*Optional[int]*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.

Returns The cluster labels for each vector in the given embedding. The vector at index n in the embedding will have the label at index n in this returned array. Noisy samples are given the value -1

Return type np.ndarray

topologic.embedding.clustering.**gaussian_mixture_model**(*embedding: numpy.ndarray*, *num_clusters: int = 1*, *seed: int = None*) → numpy.ndarray

Performs gaussian mixture model clustering on the feature_matrix.

Parameters

- **embedding** (*numpy.ndarray*) – An n x d feature matrix; it is assumed that the d features are ordered
- **num_clusters** (*int*) – How many clusters to look at between min_clusters and max_clusters, default 1
- **seed** (*Optional[int]*) – The seed for numpy random, default None

Returns The cluster labels for each vector in the given embedding. The vector at index n in the embedding will have the label at index n in this returned array

Return type np.ndarray

topologic.embedding.clustering.**kmeans**(*embedding: numpy.ndarray*, *n_clusters: int = 1*, *init: Union[str, numpy.ndarray] = 'k-means++'*, *n_init: int = 10*, *max_iter: int = 300*, *tolerance: float = 0.0001*, *precompute_distances='auto'*, *verbose: int = 0*, *random_state: int = None*, *copy_x: bool = True*, *n_jobs: int = None*, *algorithm: str = 'auto'*) → numpy.ndarray

Performs kmeans clustering on the embedding.

Parameters

- **embedding** (*numpy.ndarray*) – An n x d array of vectors representing n labels in a d dimensional space

- **n_clusters** (*int*) – The number of clusters to form as well as the number of centroids to generate. Default 1
- **init** (*Union[str, numpy.ndarray]*) – Method for initialization, defaults to 'k-means++':

  'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.

  'random': choose k observations (rows) at random from data for the initial centroids.

  If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.
- **n_init** (*int*) – Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia. Default 10
- **max_iter** (*int*) – Maximum number of iterations of the k-means algorithm for a single run. Default 300
- **tolerance** (*float*) – Relative tolerance with regards to inertia to declare convergence. Default 1e-4
- **precompute_distances** (*Union[bool, str]*) – Precompute distances (faster but takes more memory).

  'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

  True : always precompute distances

  False : never precompute distances
- **verbose** (*int*) – Verbosity mode. Default 0
- **random_state** (*Optional[Union[int, numpy.random.RandomState]]*) – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic.
- **copy_x** (*Optional[bool]*) – When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified, ensuring X is C-contiguous. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean, in this case it will also not ensure that data is C-contiguous which may cause a significant slowdown.
- **n_jobs** (*Optional[int]*) – The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

  `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **algorithm** (*str*) – K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

Returns The cluster labels for each vector in the given embedding. The vector at index n in the embedding will have the label at index n in this returned array

Return type numpy.ndarray

topologic.embedding.clustering.**wards_clustering**(*embedding: numpy.ndarray*, *num_clusters: int = 2*, *affinity: str = 'euclidean'*, *memory: str = None*, *connectivity: numpy.ndarray = None*, *compute_full_tree: str = 'auto'*) → numpy.ndarray

Uses agglomerative clustering with ward linkage

Recursively merges the pair of clusters that minimally increases a given linkage distance.

Parameters

- **embedding** (*numpy.ndarray*) – An n x d array of vectors representing n labels in a d dimensional space
- **num_clusters** (*int*) – int, default=2 The number of clusters to find.
- **affinity** (*str*) – string or callable, default: "euclidean" Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or 'precomputed'. If linkage is "ward", only "euclidean" is accepted.
- **memory** (*Optional[Union[str, joblib.Memory]]*) – None, str or object with the joblib.Memory interface, optional Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.
- **connectivity** (*numpy.ndarray*) – array-like or callable, optional Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.
- **compute_full_tree** (*Optional[str]*) – bool or 'auto' (optional) Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree.

Returns The cluster labels for each vector in the given embedding. The vector at index n in the embedding will have the label at index n in this returned array

Return type np.ndarray

topologic.embedding.distance package

topologic.embedding.distance.**cosine**(*first_vector: numpy.ndarray*, *second_vector: numpy.ndarray*) → float

Distance function for two vectors of equal length.

Cosine distance

See also: https://en.wikipedia.org/wiki/Cosine_similarity

Parameters

- **first_vector** (*numpy.ndarray*) – nonzero vector. must be same length as second_vector
- **second_vector** (*numpy.ndarray*) – nonzero vector. must be same length as first_vector

Returns cosine distance - Resulting range is between 0 and 2. Values closer to 0 are more similar. Values closer to 2 are approaching total dissimilarity.

Return type float

Examples

```
>>> cosine(np.array([1,3,5]), np.array([2,3,4]))
0.026964528109766017
```

topologic.embedding.distance.**euclidean**(*first_vector: numpy.ndarray*, *second_vector: numpy.ndarray*) → float

Distance function for two vectors of equal length

Euclidean distance

See also: https://en.wikipedia.org/wiki/Euclidean_distance

Parameters

- **first_vector** (*numpy.ndarray*) – nonzero vector. must be same length as second_vector
- **second_vector** (*numpy.ndarray*) – nonzero vector. must be same length as first_vector

Returns euclidean distance - Resulting range is a positive real number. Values closer to 0 are more similar.

Return type float

Examples

```
>>> euclidean(np.array([1,3,5]), np.array([2,3,4]))
1.4142135623730951
```

topologic.embedding.distance.**mahalanobis**(*inverse_covariance: numpy.ndarray*) → Callable[[numpy.ndarray, numpy.ndarray], float]

Unlike cosine and euclidean distances which scipy provides that take in only two vectors, mahalanobis also requires an inverse covariance matrix. This function can be used but first this matrix must be provided and a curried function handler returned, which can then be passed in to the *vector_distance* and *embedding_distances_from* functions.

See: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.mahalanobis.html

Parameters **inverse_covariance** (*np.ndarray*) – The inverse covariance matrix

Returns A curried function that now takes in 2 vectors and determines distance based on the inverse_covariance provided.

topologic.embedding.distance.**valid_distance_functions**() → KeysView[str]

The topologic builtin list of valid distance functions. Any function that return a float when given two np.ndarray 1d vectors is a valid choice, but the only ones we support without any other work are cosine or euclidean.

Returns A set-like view of the string names of the functions we support

topologic.embedding.distance.**vector_distance**(*first_vector: numpy.ndarray, second_vector: numpy.ndarray, method: Union[str, Callable[[numpy.ndarray, numpy.ndarray], float]] = <function cosine>*) → float

Vector distance is a function that will do any distance function you would like on two vectors. This is most

commonly used by changing the method parameter, as a string, from "cosine" to "euclidean" - allowing you to change your flow based on configuration not on code changes to the actual cosine and euclidean functions.

Parameters

- **first_vector** (*np.ndarray*) – A 1d array-like (list, tuple, np.array) that represents the first vector
- **second_vector** (*np.ndarray*) – A 1d array-like (list, tuple, np.array) that represents the second vector
- **method** (*Union[str, Callable[[np.ndarray, np.ndarray], float]]*) – Method can be any distance function that takes in 2 parameters. It can also be the string mapping to that function (as described by valid_distance_functions()). Note that you can also provide other functions, such as *mahalanobis*, but they require more information than just the comparative vectors.

Returns A float indicating the distance between two vectors.

topologic.embedding.distance.**embedding_distances_from**(*vector: numpy.ndarray, embedding: Union[topologic.embedding.embedding_container.Em numpy.ndarray], method: Union[str, Callable[[numpy.ndarray, numpy.ndarray], float]] = <function cosine>*) → numpy.ndarray

This function will return a 1d np.ndarray of floats by doing a distance calculation from the given *vector* to each *vector* stored in the embedding (likely including itself).

The distance calculation can be provided either as a function reference or a string representation mapped to the 2 standard distance functions we natively support. The functions supported are cosine and euclidean, both of which are *scipy* implementations. There is also a mahalanobis generator function that can be used, but first you must provide it with the inverse covariance matrix necessary for the distance calculations to be performed.

Parameters

- **vector** (*np.ndarray*) – A 1d array-like (list, tuple, np.array) that represents the vector to compare against every other vector in the embedding
- **np.ndarray] embedding** (*Union[EmbeddingContainer,*) – The embedding is either a 2d np array, where each row is a vector and the number of columns is identical to the length of the vector to compare against.
- **method** (*Union[str, Callable[[np.ndarray, np.ndarray], float]]*) – Method can be any distance function that takes in 2 parameters. It can also be the string mapping to that function (as described by valid_distance_functions()). Note that you can also provide other functions, such as *mahalanobis*, but they require more information than just the comparative vectors.

Returns np.ndarray of dtype float the same length as the count of embedded vectors

Examples

```
>>> vector = [0.3, 0.4, 0.5]
>>> embedding = np.array([[0.3, 0.4, 0.5], [0.31, 0.44, 0.7]])
>>> embedding_distances_from(vector, embedding, method="cosine") #␣
↪using string version of method name
array([0.        , 0.00861606])
```

(continues on next page)

(continued from previous page)

```
>>> embedding_distances_from(vector, embedding, method=euclidean) #␣
↪using function handle
array([0.        , 0.20420578])
```

topologic.embedding.metric package

topologic.embedding.metric.**calculate_internal_external_densities**(*graph: networkx.classes.graph.Graph*, *partitions: Dict[Any, Any]*, *weight_attribute: str = 'weight'*) → Tuple[Dict[Any, List[float]], Dict[Any, List[float]]]

Calculates the internal and external densities given a graph and a node membership dictionary. Density is defined by 'How to Make the Team: Social Networks vs. Demography as Criteria for Designing Effective Teams' as being the mean strength of tie between members of the set. In other words, density is the normalized average of edge weights by node.

For a given node, the density is the sum of all edge weights divided by the maximum edge weight for that node.

For internal density, only the edge's whose target node is in the same membership group will be summed. Similarly, for external density, only the edge's whose target node is not in the same membership group will be summed.

See also: Reagans, R., Zuckerman, E., & McEvily, B. (2004). How to Make the Team: Social Networks vs. Demography as Criteria for Designing Effective Teams. Administrative Science Quarterly, 49(1), 101–133. https://doi.org/10.2307/4131457

Parameters

- **graph** – A weighted graph that the internal density will be calculated over
- **int] partitions** (*Dict[any,*) – A dictionary for the graph with each key being a node id and each value is the membership for that node id. Often this will be a partition dictionary calculated from topologic.louvain.best_partition
- **weight_attribute** (*str*) – The key to the weight column on the graph's edges

Returns A tuple of two dictionaries. The first is the internal density and the second is the external density

Return type Tuple[Dict[Any, List[float]], Dict[Any, List[float]]]

topologic.embedding.metric.**mean_average_precision**(*graph: networkx.classes.graph.Graph*, *embedding_container: topologic.embedding.embedding_container.EmbeddingContainer*, *distance_metric: str = 'euclidean'*) → float

Mean Average Precision (mAP)

A fidelity measure to evaluate the quality of embedding generated with respect to the original unweighted Graph.

Higher mAP value corresponds to a better quality embedding.

Parameters

- **G** (*networkx.Graph*) – The unweighted graph for which the embedding is generated
- **embedding_container** (EmbeddingContainer) – The embedding container generated for the graph for which the mean average precision will be calculated
- **distance_metric** (*str*) – The distance metric to be used to find shortest path between nodes in the graph and embedding space. Default value for this param is 'euclidean', but all distance metrics available to the scipy.spatial.distance.cdist function are valid.

Returns The mean average precision (mAP <= 1) representing the quality of the embedding

Return type float

topologic.embedding.metric.**procrustes_error**(*target_matrix: numpy.ndarray*, *matrix_to_rotate: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Procrustes rotation rotates a matrix to maximum similarity with a target matrix minimizing sum of squared differences. Procrustes rotation is typically used in comparison of ordination results. It is particularly useful in comparing alternative solutions in multidimensional scaling.

For more information: https://www.rdocumentation.org/packages/vegan/versions/2.4-2/topics/procrustes

Parameters

- **target_matrix** (*numpy.ndarray*) – A matrix representing an embedding
- **matrix_to_rotate** (*numpy.ndarray*) – A matrix representing an embedding which will be rotated

Returns The error which is the difference between the two matrices and the transformation matrix

topologic.io package

topologic.io.**consolidate_bipartite**(*csv_dataset: topologic.io.datasets.CsvDataset*, *vertex_column_index: int*, *pivot_column_index: int*) → networkx.classes.graph.Graph

class topologic.io.**CsvDataset**(*source_iterator: Union[TextIO, Iterator[str]]*, *has_headers: Optional[bool] = None*, *dialect: Union[str, csv.Dialect, None] = None*, *use_headers: Optional[List[str]] = None*, *sample_size: int = 50*)

Bases: object

FIELD_SIZE_LIMIT = 2147483647

dialect() → Union[_csv.Dialect, csv.Dialect]

Note: return type information is broken due to typeshed issues with the csv module.

Returns Dialect used within this CsvDataset for the csv.reader.

Return type Union[_csv.Dialect, csv.Dialect]

headers() → List[str]

Returns Returns a *copy* of the headers.

Return type List[str]

reader() → Iterator[List[str]]

Returns Returns a properly configured csv reader for a given dialect

Return type Iterator[List[str]]

topologic.io.**find_edges**(*csv_dataset: topologic.io.datasets.CsvDataset*, *common_values_count: int = 20*, *rare_values_count: int = 20*)

topologic.io.**from_dataset**(*csv_dataset: topologic.io.datasets.CsvDataset*, *projection_function_generator: Callable[[networkx.classes.graph.Graph], Callable[[List[str]], None]]*, *graph: Optional[networkx.classes.graph.Graph] = None*) → networkx.classes.graph.Graph

Load a graph from a source csv

The most important part of this function is selecting the appropriate projection function generators. These functions generate yet another function generator, which in turn generates the function we will use to project the source CsvDataset into our graph.

The provided projection function generators fall into 3 groups:

- edges we don't want any metadata for (note that there is no vertex version of this - if you don't want vertex metadata, don't provide a vertex_csv_dataset or function!)
- edges or vertices we want metadata for, but the file is ordered sequentially and we only want the last metadata to be available in the graph
- edges or vertices we want metadata for, and we wish to keep track of every record of metadata for the edge or vertex in a list of metadata dictionaries

You can certainly provide your own projection function generators for specialized needs; just ensure they follow the type signature of Callable[[nx.Graph], Callable[[List[str]], None]]

Parameters

- **csv_dataset** (*CsvDataset*) – the dataset to read from row by row
- **projection_function_generator** (*Callable[[nx.Graph], Callable[[List[str]], None]]*) – The projection function generator function. When called with a nx.Graph, it will return the actual projection function to be used when processing each row of data.
- **graph** (*nx.Graph*) – The graph to populate. If not provided a new one is created of type nx.Graph. Note that from_dataset can be called repeatedly with different edge or vertex csv_dataset files to populate the graph more and more. If you seek to take this approach, ensure you use the same Graph object from the previous calls so that it is continuously populated with the updated data from new files

Returns the graph object

Return type nx.Graph

topologic.io.**from_file**(*edge_csv_file: TextIO*, *source_column_index: int*, *target_column_index: int*, *weight_column_index: Optional[int] = None*, *edge_csv_has_headers: Optional[bool] = None*, *edge_dialect: Union[str, csv.Dialect, None] = None*, *edge_csv_use_headers: Optional[List[str]] = None*, *edge_metadata_behavior: str = 'none'*, *edge_ignored_values: Optional[List[str]] = None*, *vertex_csv_file: Optional[TextIO] = None*, *vertex_column_index: Optional[int] = None*, *vertex_csv_has_headers: Optional[bool] = None*, *vertex_dialect: Union[str, csv.Dialect, None] = None*, *vertex_csv_use_headers: Optional[List[str]] = None*, *vertex_metadata_behavior: str = 'single'*, *vertex_ignored_values: Optional[List[str]] = None*, *sample_size: int = 50*, *is_digraph: bool = False*) → networkx.classes.graph.Graph

This function weaves a lot of graph materialization code into a single call.

The only required arguments are necessary for the bare minimum of creating a graph from an edge list. However, it is definitely recommended to specify whether the any data files use headers and a dialect; in this way we can avoid relying on the csv module's sniffing ability to detect it for us. We only use a modest number of records to discern the likelihood of headers existing or what to use for column separation (tabs or commas? quotes or double quotes? Better to specify your own dialect than hope for the best, but the capability exists if you want to throw caution to the wind.

The entire vertex metadata portion is optional; if no vertex_csv_file is specified (or it is set to None), no attempt will be made to enrich the graph node metadata. The resulting vertex_metadata_types dictionary in the NamedTuple will be an empty dictionary and can be discarded.

Likewise if no metadata is requested for projection by the edge projection function, the edge_metadata_types dictionary in the NamedTuple will be an emtpy dictionary and can be discarded.

Lastly, it is important to note that the options for edge_metadata_behavior can only be the 3 string values specified in the documentation - see the docs for that parameter for details. This is also true for the vertex_metadata_behavior - see the docs for that parameter as well.

Parameters

- **edge_csv_file** (*typing.TextIO*) – A csv file that represents the edges of a graph. This file must contain at minimum two columns: a source column and a target column. It is suggested there also exist a weight column with some form of numeric value (e.g. 30 or 30.0)
- **source_column_index** (*int*) – The column index the source vertex will be in. Columns start at 0.
- **target_column_index** (*int*) – The column index the target vertex will be in. Columns start at 0.
- **weight_column_index** (*Optional[int]*) – The column index the weight vertex will be in. Columns start at 0. If no weight_column_index is provided, we use a count of the number of VertexA to VertexB edges that exist and use that as the weight.
- **edge_csv_has_headers** (*Optional[bool]*) – Does the source CSV file contain headers? If so, we will skip the first line. If edge_csv_use_headers is a List[str], we will use those as headers for mapping any metadata. If it is None, we will use the header row as the headers, i.e. *edge_csv_use_headers* will take precedence over any headers in the source file, if applicable.
- **edge_dialect** (*Optional[Union[csv.Dialect, str]]*) – The dialect to use when parsing the source CSV file. See https://docs.python.org/3/library/csv.html#csv.Dialect for more details. If the value is None, we attempt to use the *csv* module's Sniffer class to detect which dialect to use based on a sample of the first 50 lines of the source csv file. String values can be used if you provide the strings "excel", "excel-tab", or "unix"
- **edge_csv_use_headers** (*Optional[List[str]]*) – Optional. Headers to use for the edge file either because the source file does not contain them or because you wish to override them with your own in a programmatic fashion.
- **edge_metadata_behavior** (*str*) – Dictates what extra data, aside from source, target, and weight, that we use from the provided edge list.
  - "none" brings along no metadata.
  - "single" iterates through the file from top to bottom; any edges between VertexA and VertexB that had metadata retained during edge projection will be overwritten with the newest row corresponding with VertexA and VertexB. See also: Clobbering

  - ”collection” iterates through the file from top to bottom; all new metadata detected between VertexA and VertexB is appended to the end of a list. All metadata is kept for all edges unless pruned via normal graph pruning mechanisms.

- **edge_ignored_values** (*List[str]*) – Optional. A list of strings to reject retention of during projection, e.g. “NULL” or “N/A” or “NONE”. Any attribute value found to be one of these words will be ignored.

- **vertex_csv_file** (*Optional[typing.TextIO]*) – A csv file that represents the vertices of a graph. This file should contain a column whose values correspond with the vertex ID in either the source or column field in the edges. If no edge exists for a Vertex, no metadata is retained.

  Note: If vertex_csv_file is None or not provided, <u>none</u> of the vertex_* arguments will be used.

- **vertex_column_index** (*Optional[int]*) – The column index the vertex id will be in. Columns start at 0. See note on vertex_csv_file.

- **vertex_csv_has_headers** (*Optional[bool]*) – Does the source CSV file contain headers? If so, we will skip the first line. If vertex_csv_use_headers is a List[str], we will use those as headers for mapping any metadata. If it is None, we will use the header row as the headers, i.e. *vertex_csv_use_headers* will take precedence over any header in the source file, if applicable.

- **str]] vertex_dialect** (*Optional[Union[csv.Dialect,*) – The dialect to use when parsing the source CSV file. See https://docs.python.org/3/library/csv.html#csv.Dialect for more details. If the value is None, we attempt to use the *csv* module’s Sniffer class to detect which dialect to use based on a sample of the first 50 lines of the source csv file. String values can be used if you provide the strings “excel”, “excel-tab”, or “unix”

- **vertex_csv_use_headers** (*Optional[List[str]]*) – Optional. Headers to use for the vertex file either because the source file does not contain them or because you wish to override them with your own in a programmatic fashion. See note on vertex_csv_file.

- **vertex_metadata_behavior** (*str*) – Dictates what we do with vertex metadata. Unlike edge metadata, there is no need to provide a vertex_metadata_behavior if you have no vertex metadata you wish to capture. No metadata will be stored for any vertex if it is not detected in the graph already; if there are no edges to or from VertexA, there will be no metadata retained for VertexA.

  - ”simple” iterates through the file from top to bottom; any vertex that had already captured metadata through the vertex metadata projection process will be overwritten with the newest metadata corresponding with that vertex.

  - ”collection” iterates through the file from top to bottom; all new metadata detected for a given vertex will

- **vertex_ignored_values** (*List[str]*) – Optional. A list of strings to reject retention of during projection, e.g. “NULL” be appended to a metadata list. or “N/A” or “NONE”. Any attribute value found to be one of these words will be ignored.

- **sample_size** (*int*) – The sample size to extract from the source CSV for use in Sniffing dialect or has_headers. Please note that this sample_size does NOT advance your underlying iterator, nor is there any guarantee that the csv Sniffer class will use every row extracted via sample_size. Setting this above 50 may not have the impact you hope for due to the csv.Sniffer.has_header function - it will use at most 20 rows.

- **is_digraph** (*bool*) – If the data represents an undirected graph or a directed graph. Default is *False*.

Returns The graph populated graph

Return type nx.Graph

class topologic.io.**GraphProperties**(*column_names*, *potential_edge_column_pairs*, *common_column_values*, *rare_column_values*)

Bases: object

column_names()

common_column_values()

Dictionary of column name to set of common values for that column and their counts

potential_edge_column_pairs()

rare_column_values()

Dictionary of column name to set of rare values for that column and their counts

topologic.io.**load**(*edge_file: str*, *separator: str = 'excel'*, *has_header: bool = True*, *source_index: int = 0*, *target_index: int = 1*, *weight_index: Optional[int] = None*) → networkx.classes.graph.Graph

Spartan, on-rails function to load an edge file.

Parameters

- **edge_file** (*str*) – String path to an edge file on the filesystem
- **separator** (*str*) – Valid values are 'excel' or 'excel-tab'.
- **has_header** (*bool*) – True if the edge file has a header line, False if not
- **source_index** (*int*) – The column index for the source vertex (default 0)
- **target_index** (*int*) – The column index for the target vertex (default 1)
- **weight_index** (*Optional[int]*) – The column index for the edge weight (default None). If None, or if there is no column at *weight_index*, weights per edge are defaulted to 1.

Returns

class topologic.io.**PotentialEdgeColumnPair**(*source*, *destination*, *score*)

Bases: object

destination()

score()

source()

topologic.io.**tensor_projection_reader**(*embedding_file_path: str*, *label_file_path: str*) → Tuple[numpy.ndarray, List[List[str]]]

Reads the embedding and labels stored at the given paths and returns an np.ndarray and list of labels

Parameters

- **embedding_file_path** (*str*) – Path to the embedding file
- **label_file_path** (*str*) – Path to the labels file

Returns An embedding and list of labels

Return type (numpy.ndarray, List[List[str]])

topologic.io.**tensor_projection_writer**(*embedding_file_path: str, label_file_path: str, vectors: numpy.ndarray, labels: Union[List[List[str]], List[str]], encoding: str = 'utf-8'*)

Writes an embedding and labels to a given vector file path and label file path in a form that Tensorboard embedding projector can read

Parameters

- **embedding_file_path** (*str*) – Path that the embedding file will be written
- **label_file_path** (*str*) – Path that the label file will be written
- **vectors** (*numpy.ndarray*) – A embedding represented as a np.ndarray
- **labels** (*Union[List[List[str]], List[str]]*) – A list of lists where each inner list is the data for a single row in the embedding or a list of strings where each string is the single label for that tensor. If you pass in a List[List[str]] you allow multiple labels for a single tensor.
- **encoding** (*str*) – The encoding used to write the file

topologic.partition package

topologic.partition.**induce_graph_by_communities**(*graph: networkx.classes.graph.Graph, communities: Dict[Any, int], weight_attribute: str = 'weight'*) → networkx.classes.graph.Graph

Creates a community graph with nodes from the communities dictionary and using the edges of the original graph to form edges between communities.

Weights are aggregated; you may need to normalize the resulting graph after calling this function.

Note: logs a warning if the size of the community dictionary is less than the size of the provided graph's vertexset.

Parameters

- **graph** (*networkx.Graph*) – The original graph that contains the edges that will be used to formulate a new induced community graph
- **communities** (*dict[Any, int]*) – The communities dictionary provides a mapping of original vertex ID to new community ID.
- **weight_attribute** (*str*) – The weight attribute on the original graph's edges to use when aggregating the weights of the induced community graph. Default is *weight*.

Returns The induced community graph.

Return type networkx.Graph

Raises

- **ValueError** – If the graph is None
- **ValueError** – If the communities dictionary is None

topologic.partition.**louvain**(*graph: networkx.classes.graph.Graph, partition: Optional[Dict[Any, int]] = None, weight_attribute: str = 'weight', resolution: float = 1.0, random_state: Any = None*) → Dict[Any, int]

Compute the partition of the graph nodes which maximises the modularity (or try..) using the Louvain heuristics

This is the partition of highest modularity, i.e. the highest partition of the dendrogram generated by the Louvain algorithm.

This louvain function is a limited wrapper to the community.best_partition function in the python-louvain library written by Thomas Aynaud.

Parameters

- **graph** (*networkx.Graph*) – the networkx graph which is decomposed
- **partition** (*Optional[Dict[Any, int]]*) – the algorithm will start using this partition of the nodes. It's a dictionary where keys are their nodes and values the communities
- **weight_attribute** (*str*) – the key in graph to use as weight. Default to 'weight'
- **resolution** (*float*) – Will change the size of the communities, default to 1. represents the time described in "Laplacian Dynamics and Multiscale Modular Structure in Networks", R. Lambiotte, J.-C. Delvenne, M. Barahona
- **random_state** (*Any*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Returns The partition, with communities numbered from 0 to number of communities

Return type Dict[Any, int]

Raises NetworkXError - If the graph is not Eulerian.

References 1. Blondel, V.D. et al. Fast unfolding of communities in large networks. J. Stat. Mech 10008, 1-12(2008).

topologic.partition.**modularity**(*graph: networkx.classes.graph.Graph, partitions: Dict[Any, int], weight_attribute: str = 'weight', resolution: float = 1.0*) → float

Given an undirected graph and a dictionary of vertices to community ids, calculate the modularity.

See also: https://en.wikipedia.org/wiki/Modularity_(networks)

Parameters

- **graph** (*nx.Graph*) – An undirected graph
- **int] partitions** (*Dict[Any,*) – A dictionary representing a community partitioning scheme with the keys being the vertex and the value being a community id. Within topologic, these community ids are required to be ints.
- **weight_attribute** (*str*) – The edge data attribute on the graph that contains a float weight for the edge.
- **resolution** (*float*) – The resolution to use when calculating the modularity.

Returns The modularity quality score for the given network and community partition schema.

Raises

- **TypeError** – If the graph is not a networkx Graph
- **ValueError** – If the graph is unweighted
- **ValueError** – If the graph is directed

topologic.partition.**modularity_components**(*graph: networkx.classes.graph.Graph, partitions: Dict[Any, int], weight_attribute: str = 'weight', resolution: float = 1.0*) → Dict[int, float]

Given an undirected, weighted graph and a community partition dictionary, calculates a modularity quantum for each community ID. The sum of these quanta is the modularity of the graph and partitions provided.

Parameters

- **graph** (*nx.Graph*) – An undirected graph
- **int] partitions** (*Dict[Any,*) – A dictionary representing a community partitioning scheme with the keys being the vertex and the value being a community id. Within topologic, these community ids are required to be ints.
- **weight_attribute** (*str*) – The edge data attribute on the graph that contains a float weight for the edge.
- **resolution** (*float*) – The resolution to use when calculating the modularity.

Returns A dictionary of the community id to the modularity component of that community

Return type Dict[int, float]

Raises

- **TypeError** – If the graph is not a networkx Graph
- **ValueError** – If the graph is unweighted
- **ValueError** – If the graph is directed

topologic.partition.**q_score**(*partitioned_graph: topologic.partitioned_graph.PartitionedGraph*, *weight_column: str = 'weight'*) → float

Deprecated: See modularity() for replacement.

Given a topologic PartitionedGraph, return the q score - or modularity of a graph.

See also: https://en.wikipedia.org/wiki/Modularity_(networks)

Parameters

- **partitioned_graph** (*Optional[topologic.PartitionedGraph]*) – Partitioned graph contains a dictionary of all the communities in a graph, optimized for best modularity. This partition structure is used when computing final q_score / modularity of graph.
- **weight_column** (*str*) – weight column to use in computing modularity.

Raises

- ***UnweightedGraphError*** – if graph does not contain weight_column in edge attributes
- **KeyError** – If the partition is not a partition of all graph nodes. This should not occur if PartitionedGraph is recently created and no changes occurred to the underlying networkx.Graph object.
- **ValueError** – If the graph has no links.
- **TypeError** – If partitioned_graph is not of type topologic.PartitionedGraph

Returns q_score, or modularity, of this graph using the provided partitioning scheme.

Return type float

topologic.projection package

topologic.projections provides a canned series of source-to-Graph projection functions.

The function return type Callable[[nx.Graph], Callable[[List[str]], None]] is the cornerstone to every projection function.

You can, and should, define any function you want as long as it complies with that return type. The first function should take in any Dataset source-specific configuration information; information necessary to tell a projection function how your source data is laid out and should be used to apply changes to the networkx.Graph.

The 1st inner function is used by the topologic.io.csv_loader.from_dataset function - it applies the networkx.Graph to update.

The 2nd inner function is also used by topologic.io.csv_loader.from_dataset to take each record from the Dataset source and use it to apply this modification to the networkx.Graph.

Graph metadata format:

If a graph is to have metadata on its vertices or edges, it shall always be in the form of:

```
{
    "weight": 1.0,
    "attributes": [
        { "key": "value1", "anotherKey": "anotherValue", "aDifferentKey":
↪"aDifferentValue" },
        { "key": "value2", "aDifferentKey": "aDifferentValue2" }
    ]
}
```

Notes: - Keys are always of type str - Values are always stored as type str, but could be a more narrowly bounded type like int or float. - Just because a key exists in one row of the attributes List does not mean it will exist in any other. Do not presume a constant "shape" of the dictionaries of edge attributes.

topologic.projection.**edge_ignore_metadata**(*source_index: int*, *target_index: int*, *weight_index: Optional[int] = None*) → Callable[[networkx.classes.graph.Graph], Callable[[List[str]], None]]

Drops all metadata. Creates graph solely based on source, target, and optional weight.

See package docstrings for more details on these currying functions.

Parameters

- **source_index** (*int*) – The index in the CSV data row to use as the source node in this edge
- **target_index** (*int*) – The index in the CSV data row to use as the target node in this edge
- **weight_index** (*Optional[int]*) – Optional. The index in the CSV data row to use as the weight of the edge. If no weight is provided, all records of an edge are presumed to have a weight of 1. Duplicates of an edge will have their weights (or inferred weight) aggregated into a single value.

Returns A partially applied function that partially applies yet more arguments prior to the final operation function

Return type Callable[[networkx.Graph], Callable[[List[str]], None]]

topologic.projection.**edge_with_collection_metadata**(*headers: List[str], source_index: int, target_index: int, weight_index: Optional[int] = None, ignored_values: Optional[List[str]] = None*) → Callable[[networkx.classes.graph.Graph], Callable[[List[str]], None]]

Some graph algorithms have undefined behavior over multigraphs. To skirt this limitation, we allow the data to represent a multigraph, though we collapse it into a non-multigraph. We do this by aggregating the weights, and in this case we take any extra metadata in the edge source and project it, along with headers, into an attribute dictionary. This dictionary is then added to a List of any previous attribute dictionaries for the same source and target, so as to not clobber any metadata.

See package docstrings for more details on these currying functions.

Parameters

- **headers** (*List[str]*) – Headers from a CSV row to use as metadata attribute keys
- **source_index** (*int*) – The index in the CSV data row to use as the source node in this edge
- **target_index** (*int*) – The index in the CSV data row to use as the target node in this edge
- **weight_index** (*Optional[int]*) – Optional. The index in the CSV data row to use as the weight of the edge. If no weight is provided, all records of an edge are presumed to have a weight of 1. Duplicates of an edge will have their weights (or inferred weight) aggregated into a single value.
- **ignored_values** (*Optional[List[str]]*) – Optional. A list of values to ignore if present in the row, such as "NULL" or ""

Returns A partially applied function that partially applies yet more arguments prior to the final operation function

Return type Callable[[networkx.Graph], Callable[[List[str]], None]]

topologic.projection.**edge_with_single_metadata**(*headers: List[str], source_index: int, target_index: int, weight_index: Optional[int] = None, ignored_values: Optional[List[str]] = None*) → Callable[[networkx.classes.graph.Graph], Callable[[List[str]], None]]

Will load edges into graph even if they are a multigraph. However, aside from weight, the multigraph attributes are ignored and the last record to be processed for that source and target will have its metadata retained and all prior metadata dropped.

See package docstrings for more details on these currying functions.

Parameters

- **headers** (*List[str]*) – Headers from a CSV row to use as metadata attribute keys
- **source_index** (*int*) – The index in the CSV data row to use as the source node in this edge
- **target_index** (*int*) – The index in the CSV data row to use as the target node in this edge
- **weight_index** (*Optional[int]*) – Optional. The index in the CSV data row to use as the weight of the edge. If no weight is provided, all records of an edge are presumed

to have a weight of 1. Duplicates of an edge will have their weights (or inferred weight) aggregated into a single value.

- **ignored_values** (*Optional[List[str]]*) – Optional. A list of values to ignore if present in the row, such as "NULL" or ""

Returns A partially applied function that partially applies yet more arguments prior to the final operation function

Return type Callable[[networkx.Graph], Callable[[List[str]], None]]

topologic.projection.**vertex_with_collection_metadata**(*headers: List[str], vertex_id_index: int, ignored_values: Optional[List[str]] = None*) → Callable[[networkx.classes.graph.Graph], Callable[[List[str]], None]]

This function is an unlikely function to use; if you have vertex metadata you wish to load into the networkx.Graph where the vertex_id is repeated, it would be a better choice for you to compact those into a single record with a specific, string representable format of multiple metadata entries. However, this function can be used when you aren't sure what you have. Like the edge_with_collection_metadata projection, this function will create a List of dictionaries for each instance of metadata it sees for a given vertex_id.

Note: If the vertex_id for a given row does not exist in the graph, NO METADATA WILL BE RETAINED.

See package docstrings for more details on these currying functions.

Parameters

- **headers** (*List[str]*) – Headers from a CSV row to use as metadata attribute keys
- **vertex_id_index** (*int*) – The index in the CSV data row to use as the vertex id in this graph
- **ignored_values** (*Optional[List[str]]*) – Optional. A list of values to ignore if present in the row, such as "NULL" or ""

Returns A partially applied function that partially applies yet more arguments prior to the final operation function

Return type Callable[[networkx.Graph], Callable[[List[str]], None]]

topologic.projection.**vertex_with_single_metadata**(*headers: List[str], vertex_id_index: int, ignored_values: List[str] = None*) → Callable[[networkx.classes.graph.Graph], Callable[[List[str]], None]]

Function will project vertex metadata into the graph. If prior data exists for the vertex_id, the later instance of data for the vertex_id will clobber it.

Note: If the vertex_id for a given row does not exist in the graph, NO METADATA WILL BE RETAINED.

See package docstrings for more details on these currying functions and on the attributes datastructure.

Parameters

- **headers** (*List[str]*) – Headers from a CSV row to use as metadata attribute keys
- **vertex_id_index** (*int*) – The index in the CSV data row to use as the vertex id in this graph
- **ignored_values** (*Optional[List[str]]*) – Optional. A list of values to ignore if present in the row, such as "NULL" or ""

Returns A partially applied function that partially applies yet more arguments prior to the final operation function

Return type Callable[[networkx.Graph], Callable[[List[str]], None]]

topologic.similarity package

topologic.similarity.**ari**(*primary_partition: Dict[Any, int], test_partition: Dict[Any, int]*) → float

Given two partition schemas, a primary partition mapping (the most accurate representation of truth) and the test partition mapping (to be scored against that accurate representation of truth), calculate the Adjusted Rand Index.

See https://en.wikipedia.org/wiki/Rand_index

Parameters

- **int] primary_partition** (*Dict[Any,*) – The most accurate representation of truth for cluster or community membership of nodes. The keys are vertex labels and the values are the cluster/community/partition labels.
- **int] test_partition** (*Dict[Any,*) – The partition mapping to compare against the primary partition. The keys are vertex labels and the values are the cluster/community/partition labels.

Returns The adjusted rand index for the two mappings

Rtype float

Raises **ValueError** – If the primary partition and test partition do not have an identical vertex label set.

topologic.statistics package

topologic.statistics.**cut_edges_by_weight**(*graph: networkx.classes.graph.Graph, cut_threshold: Union[int, float], cut_process: topologic.statistics.make_cuts.MakeCuts, weight_attribute: str = 'weight', prune_isolates: bool = False*) → networkx.classes.graph.Graph

Given a graph, a cut threshold, and a cut_process, create a new Graph that contains only the edges that are not pruned.

Note: Edges without a *weight_attribute* field will be excluded from these cuts. Enable logging to view any messages about edges without weights.

Parameters

- **graph** (*networkx.Graph*) – The graph that will be copied and pruned.
- **cut_threshold** (*Union[int, float]*) – The threshold for making cuts based on weight.
- **cut_process** (MakeCuts) – Describes how we should make the cut; cut all edges larger or smaller than the cut_threshold, and whether exclusive or inclusive.
- **weight_attribute** (*str*) – The weight attribute name in the data dictionary. Default is *weight*.
- **prune_isolates** (*bool*) – If true, remove any vertex that no longer has an edge. Note that this only prunes vertices which have edges to be pruned; any isolate vertex prior to any edge cut will be retained.

Returns Pruned copy of the graph

Return type networkx.Graph

topologic.statistics.**cut_vertices_by_betweenness_centrality**(*graph: networkx.classes.graph.Graph, cut_threshold: Union[int, float], cut_process: topologic.statistics.make_cuts.MakeCuts, num_random_samples: Optional[int] = None, normalized: bool = True, weight_attribute: Optional[str] = None, include_endpoints: bool = False, random_seed: Union[int, random.Random, None] = None*) → networkx.classes.graph.Graph

Given a graph and a cut_threshold and a cut_process, return a copy of the graph with the vertices outside of the cut_threshold.

The betweenness centrality calculation can take advantage of networkx' implementation of randomized sampling by providing num_random_samples (or k, in networkx betweenness_centrality nomenclature).

See: https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html for more details.

Parameters

- **graph** (*networkx.Graph*) – The graph that will be copied and pruned.
- **cut_threshold** (*Union[int, float]*) – The threshold for making cuts based on betweenness centrality.
- **cut_process** (MakeCuts) – Describes how we should make the cut; cut all edges larger or smaller than the cut_threshold, and whether exclusive or inclusive.
- **num_random_samples** (*Optional[int]*) – Use num_random_samples for vertex samples to *estimate* betweenness. num_random_samples should be <= len(graph.nodes). The larger num_random_samples is, the better the approximation.
- **normalized** (*bool*) – If True the betweenness values are normalized by 2/((n-1)(n-2)) for graphs, and 1/((n-1)(n-2)) for directed graphs where n is the number of vertices in the graph.
- **weight_attribute** (*Optional[str]*) – If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.
- **include_endpoints** (*bool*) – If True include the endpoints in the shortest path counts.
- **random_seed** (*Optional[Union[int, random.Random]]*) – Random seed or preconfigured random instance to be used for randomly selecting random samples. Only used if num_random_samples is set. None will generate a new random state. Specifying a random state will provide consistent results between runs.

Returns Pruned copy of the graph

Return type networkx.Graph

topologic.statistics.**cut_vertices_by_degree_centrality**(*graph: networkx.classes.graph.Graph, cut_threshold: Union[int, float], cut_process: topologic.statistics.make_cuts.MakeCuts*) → networkx.classes.graph.Graph

Given a graph and a cut_threshold and a cut_process, return a copy of the graph with the vertices outside of the cut_threshold.

Parameters

- **graph** (*networkx.Graph*) – The graph that will be copied and pruned.
- **cut_threshold** (*Union[int, float]*) – The threshold for making cuts based on degree centrality.
- **cut_process** (MakeCuts) – Describes how we should make the cut; cut all edges larger or smaller than the cut_threshold, and whether exclusive or inclusive.

Returns Pruned copy of the graph

Return type networkx.Graph

class topologic.statistics.**DefinedHistogram**

Bases: tuple

Contains the histogram and the edges of the bins in the histogram.

The bin_edges will have a length 1 greater than the histogram, as it defines the minimal and maximal edges as well as each edge in between.

property bin_edges

Alias for field number 1

property histogram

Alias for field number 0

topologic.statistics.**filter_function_for_make_cuts**(*cut_threshold: Union[int, float], cut_process: topologic.statistics.make_cuts.MakeCuts*) → Callable[[Tuple[Any, Union[int, float]]], bool]

topologic.statistics.**histogram_betweenness_centrality**(*graph: networkx.classes.graph.Graph, bin_directive: Union[int, List[Union[float, int]], numpy.ndarray, str] = 10, num_random_samples: Optional[int] = None, normalized: bool = True, weight_attribute: Optional[str] = None, include_endpoints: bool = False, random_seed: Union[int, random.Random, None] = None*) → topologic.statistics.defined_histogram.DefinedHistogram

Generates a histogram of the vertex betweenness centrality of the provided graph. Histogram function is fundamentally proxied through to numpy's *histogram* function, and bin selection follows *numpy.histogram* processes.

The betweenness centrality calculation can take advantage of networkx' implementation of randomized sampling by providing num_random_samples (or k, in networkx betweenness_centrality nomenclature).

See: https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html for more details.

Parameters

- **graph** (*networkx.Graph*) – the graph. No changes will be made to it.
- **bin_directive** (*Union[int, List[Union[float, int]], numpy.ndarray, str]*) – Is passed directly through to numpy's "histogram" (and thus, "histogram_bin_edges") functions. See: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.histogram_bin_edges.html#numpy.histogram_bin_edges In short description: if an int is provided, we use *bin_directive* number of equal range bins. If a sequence is provided, these bin edges will be used and can be sized to whatever size you prefer. Note that the np.ndarray should be ndim=1 and the values should be float or int.
- **num_random_samples** (*Optional[int]*) – Use num_random_samples for vertex samples to *estimate* betweeness. num_random_samples should be <= len(graph.nodes). The larger num_random_samples is, the better the approximation.
- **normalized** (*bool*) – If True the betweenness values are normalized by 2/((n-1)(n-2)) for graphs, and 1/((n-1)(n-2)) for directed graphs where n is the number of vertices in the graph.
- **weight_attribute** (*Optional[str]*) – If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.
- **include_endpoints** (*bool*) – If True include the endpoints in the shortest path counts.
- **random_seed** (*Optional[Union[int, random.Random]]*) – Random seed or preconfigured random instance to be used for randomly selecting random samples. Only used if num_random_samples is set. None will generate a new random state. Specifying a random state will provide consistent results between runs.

Returns A named tuple that contains the histogram and the bin_edges used in the histogram

Return type *DefinedHistogram*

topologic.statistics.**histogram_degree_centrality**(*graph: networkx.classes.graph.Graph, bin_directive: Union[int, List[Union[float, int]], numpy.ndarray, str] = 10*) → topologic.statistics.defined_histogram.DefinedHistogram

Generates a histogram of the vertex degree centrality of the provided graph. Histogram function is fundamentally proxied through to numpy's *histogram* function, and bin selection follows *numpy.histogram* processes.

Parameters

- **graph** (*networkx.Graph*) – the graph. No changes will be made to it.
- **bin_directive** (*Union[int, List[Union[float, int]], numpy.ndarray, str]*) – Is passed directly through to numpy's "histogram" (and thus, "histogram_bin_edges") functions. See: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.histogram_bin_edges.html#numpy.histogram_bin_edges In short description: if an int is provided, we use *bin_directive* number of equal range bins. If a sequence is provided, these bin edges will be used and can be sized to whatever size you prefer Note that the np.ndarray should be ndim=1 and the values should be float or int.

Returns A named tuple that contains the histogram and the bin_edges used in the histogram

Return type *DefinedHistogram*

topologic.statistics.**histogram_edge_weight**(*graph: networkx.classes.graph.Graph, bin_directive: Union[int, List[Union[float, int]], numpy.ndarray, str] = 10, weight_attribute: str = 'weight'*) → topologic.statistics.defined_histogram.DefinedHistogram

Generates a histogram of the edge weights of the provided graph. Histogram function is fundamentally proxied through to numpy's *histogram* function, and bin selection follows *numpy.histogram* processes.

Note: Edges without a *weight_attribute* field will be excluded from this histogram. Enable logging to view any messages about edges without weights.

Parameters

- **graph** (*networkx.Graph*) – the graph. No changes will be made to it.
- **bin_directive** (*Union[int, List[Union[float, int]], numpy.ndarray, str]*) – Is passed directly through to numpy's "histogram" (and thus, "histogram_bin_edges") functions. See: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.histogram_bin_edges.html#numpy.histogram_bin_edges In short description: if an int is provided, we use *bin_directive* number of equal range bins. If a sequence is provided, these bin edges will be used and can be sized to whatever size you prefer Note that the np.ndarray should be ndim=1 and the values should be float or int.
- **weight_attribute** (*str*) – The weight attribute name in the data dictionary. Default is *weight*.

Returns A named tuple that contains the histogram and the bin_edges used in the histogram

Return type *DefinedHistogram*

class topologic.statistics.**MakeCuts**

Bases: enum.Enum

An enumeration.

LARGER_THAN_EXCLUSIVE = 2

LARGER_THAN_INCLUSIVE = 1

```
SMALLER_THAN_EXCLUSIVE = 4
SMALLER_THAN_INCLUSIVE = 3
```

CHAPTER

TWO

SYSTEM REQUIREMENTS

`topologic` is written for Python 3.6. It is well tested under Python 3.7 and may work well with Python 3.8. It makes use of type hinting heavily, so it is not likely to work with Python 3.5.

In addition, some of the library dependencies for `topologic` must be built on your system, and will require C++ build tools to complete. If you don't already have these, the install process will fail, and you can try some of the following steps to fix your issues.

2.1 Windows

Visit Visual Studio and select the `Tools for Visual Studio 2017` header. Then download and install the `Build Tools for Visual Studio 2017`.

2.2 Ubuntu Linux

If using Python3.6:

```
sudo apt install build-essential python3.6-dev
```

If using Python3.7:

```
sudo apt install build-essential python3.7-dev
```

CHAPTER

THREE

RELEASE NOTES

3.1 0.1.8

- Fix an issue with sorting non integer node ids when calculating omnibus embeddings

3.2 0.1.7

- Use the union graph largest connected component strategy to calculate the omnibus embedding

3.3 0.1.6

- Fix an issue that caused inf and nan when using LSE omnibus embedding

3.4 0.1.5

- Fix a bug in omnibus embedding where augmentation happened before the graphs were reduced to common nodes

3.5 0.1.4

- Fixed a bug during Laplacian matrix construction for directed graphs

3.6 0.1.3

- Added `modularity` and `modularity_components` functions, and deprecated `q_score`.

3.7 0.1.2

- Rename `self_loop_augmentation` to `diagonal_augmentation` and use weighted degree to perform calculation instead of degree only.
- Fix bug when getting the length of edges when performing graph augmentations.

3.8 0.1.1

- Issue 29 Fixed bug in `topologic.io.from_dataset` where an empty networkx graph object (Graph, DiGraph, etc) was being treated as if no networkx Graph object were provided at all.
- Added `is_digraph` parameter to `topologic.io.from_file`. This parameter defaults to False for original behavior. Setting it to True will create a networkx DiGraph object instead.

3.9 0.1.0

- Initial release

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

U

V

W