Netseer

Predicting the Graph Structure

None

None

Table of contents

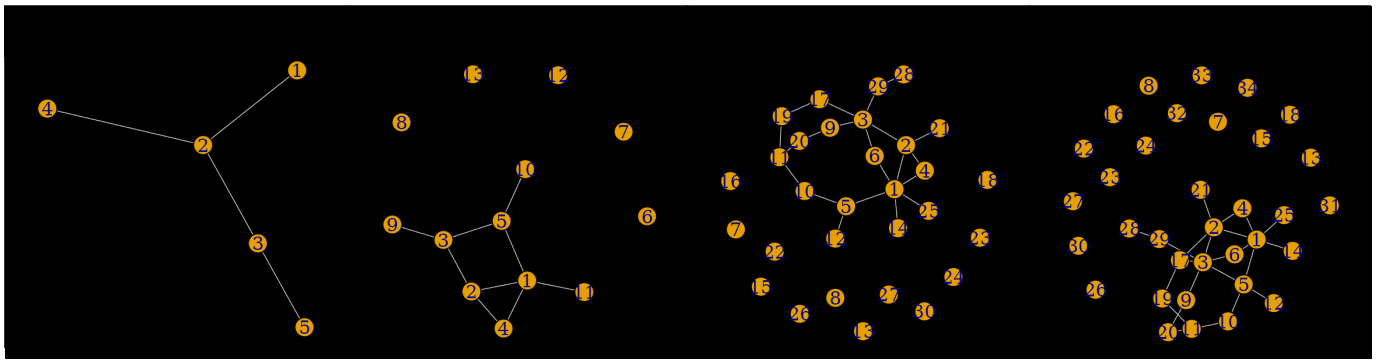
1. Netseer

1.1 Predicting graph structure from a time series of graphs

This is a Python implementation of *netseer*.
Netseer is a tool that outputs a predicted graph based on a time series graph sequence

1.2 Purpose

The goal of netseer is to predict the graph structure including new nodes and edges from a time series of graphs.
The methodology is explained in the preprint (Kandanaarachchi et al. 2025).



1.3 Authors

Stefan Westerlund: Created netseer code.
Brodie Oldfield: Packaging and docs.

1.4 Installation

This package is available on PyPI, and can be installed with PIP or with a Package Manager:

```
pip install netseer # or uv add netseer
```

1.5 Quick Example

Generating an example graph list:

```
import netseer as ns

graph_list = ns.generate_graph_linear(num_iters = 15)
```

The `generate_graph_list()` function has parameters for templating what types of graphs to generate. Information about these can be found in the reference docs.
The `num_iters` parameter sets how many graphs to generate.

Predicting on that graph:

```
predict = ns.predict_graph(graph_list, h=1)
```

Increasing the `h` parameter increases how many steps into the future the prediction is, with `h=1` being 1 step in the graph sequence.

After creating a new predicted graph, you can compare it to the original graph list.
As the original graph list has 15 graphs, we can compare the 15th original graph to the predicted 16th graph.

```
vertex_error, edge_error = ns.measure_error(graph_list[14], predict)
print(vertex_error)
print(edge_error)
```

2. References

Contains functions for generating a list of graphs.

2.1 `generate_graph_exp(start_nodes=1000, add_nodes=50, add_edges=200, rem_edges=100, nn_edges=3, num_iters=15)`

TODO: Not Implemented Randomly generates an exponentially growing list of graphs using supplied parameters.

Parameters:

Name	Type	Description	Default
`start_nodes`	`int`	Number of nodes the first graph will have.	`1000`
`add_nodes`	`int`	Number of nodes the graph will grow by each step.	`50`
`add_edges`	`int`	Number of edges the graph will grow by each step.	`200`
`rem_edges`	`int`	Number of edges that are removed each step.	`100`
`num_iters`	`int`	How many times should the graph grow.	`15`

Returns:

Type	Description
`list[Graph]`	A list of graphs, with index 0 being the first generated graph.

Source code in `src/netseer/graph_generation.py`

```python
def generate_graph_exp(
    start_nodes: int = 1000,
    add_nodes: int = 50,
    add_edges: int = 200,
    rem_edges: int = 100,
    nn_edges: int = 3,
    num_iters: int = 15,
) -> list[ig.Graph]:
    """TODO: Not Implemented
    Randomly generates an exponentially growing list of graphs using supplied parameters.

    Args:
        start_nodes: Number of nodes the first graph will have.
        add_nodes: Number of nodes the graph will grow by each step.
        add_edges: Number of edges the graph will grow by each step.
        rem_edges: Number of edges that are removed each step.
        num_iters: How many times should the graph grow.

    Returns:
        A list of graphs, with index 0 being the first generated graph.
    """
    graph_list = [None] * num_iters
    initial_graph = ig.Graph.Barabasi(n=start_nodes, directed=False)
    graph_list[0] = initial_graph
    # TODO: Implement Exp Growth.
    for i in range(1, num_iters):
        graph_list[i] = generate_next_graph(
            graph_list[i - 1], add_nodes, add_edges, rem_edges, nn_edges
        )
    return graph_list
```

2.2 `generate_graph_linear(start_nodes=1000, add_nodes=50, add_edges=200, rem_edges=100, nn_edges=3, num_iters=15)`

Randomly generates a linearly growing list of graphs using supplied parameters.

Parameters:

Name	Type	Description	Default
`start_nodes`	`int`	Number of nodes the first graph will have.	`1000`
`add_nodes`	`int`	Number of nodes the graph will grow by each step.	`50`
`add_edges`	`int`	Number of edges the graph will grow by each step.	`200`
`rem_edges`	`int`	Number of edges that are removed each step.	`100`
`num_iters`	`int`	How many times should the graph grow.	`15`

Returns:

Type	Description
`list[Graph]`	A list of graphs, with index 0 being the first generated graph.

Source code in `src/netseer/graph_generation.py`

```python
def generate_graph_linear(
    start_nodes: int = 1000,
    add_nodes: int = 50,
    add_edges: int = 200,
    rem_edges: int = 100,
    nn_edges: int = 3,
    num_iters: int = 15,
) -> list[ig.Graph]:
    """Randomly generates a linearly growing list of graphs using supplied parameters.

    Args:
        start_nodes: Number of nodes the first graph will have.
        add_nodes: Number of nodes the graph will grow by each step.
        add_edges: Number of edges the graph will grow by each step.
        rem_edges: Number of edges that are removed each step.
        num_iters: How many times should the graph grow.

    Returns:
        A list of graphs, with index 0 being the first generated graph.
    """
    graph_list = [None] * num_iters
    initial_graph = ig.Graph.Barabasi(n=start_nodes, directed=False)
    graph_list[0] = initial_graph
    for i in range(1, num_iters):
        graph_list[i] = generate_next_graph(
            graph_list[i - 1], add_nodes, add_edges, rem_edges, nn_edges
        )
    return graph_list
```

Prediction Module:

2.3 `predict_graph(graph_list, h=5, conf_nodes=None, conf_degree=90, weights_option=4, weights_param=0.001)`

From a list of graphs, predicts a new graph

Parameters:

Name	Type	Description	Default
`graph_list`	`Iterable[Graph]`	A list of graphs to be predicted on: See generate_graph_list()	*required*
`h`	`int`	How many steps into the future the prediction should be.	`5`
`conf_nodes`		-- Not Implemented.	`None`
`conf_degree`	`int`	-- Not Implemented.	`90`
`weights_option`	`int`	Int between 1-7 determines edge weight schemes. 1: Uniform Weight. 1 for all Edges. 2: Binary Weight. 1 for all Edges. 3: Binary Weight. 1 for most connected vertices. 4: Proportional Weights according to history. 5: Proportional Weight. Linear scaling weights for new edges based on time series (Time Step Index+ 1). E.g. 1, 2, 3 6: Proportional Weights. Decaying weights based on time series. (1/(Number of graphs - Time Step Index + 1)) E.g. 1, 1/2, 1/3 7: Weight 0 for all edges, except for last edge which is 1. 8: Not Implemented.	`4`
`weights_param`		The weight given for possible edges from new vertices. Default is 0.001	`0.001`

Returns:

Type	Description
`Graph`	Predicted Graph.

Source code in `src/netseer/network_prediction.py`

```python
def predict_graph(
    graph_list: Iterable[ig.Graph],
    h: int = 5,
    conf_nodes=None,
    conf_degree: int = 90,
    weights_option: int = 4,
    weights_param=0.001,
) -> ig.Graph:
    """From a list of graphs, predicts a new graph

    Args:
        graph_list: A list of graphs to be predicted on: See generate_graph_list()
        h: How many steps into the future the prediction should be.
        conf_nodes: -- Not Implemented.
        conf_degree: -- Not Implemented.
        weights_option: Int between 1-7 determines edge weight schemes.
            1: Uniform Weight. 1 for all Edges.
            2: Binary Weight. 1 for all Edges.
            3: Binary Weight. 1 for most connected vertices.
            4: Proportional Weights according to history.
            5: Proportional Weight. Linear scaling weights for new edges based on time series (Time Step Index+ 1 ). E.g. 1, 2, 3
            6: Proportional Weights. Decaying weights based on time series. (1/(Number of graphs - Time Step Index + 1)) E.g. 1, 1/2, 1/3
            7: Weight 0 for all edges, except for last edge which is 1.
            8: Not Implemented.
        weights_param: The weight given for possible edges from new vertices. Default is 0.001

    Returns:
        Predicted Graph.
    """
    if not 1 <= weights_option <= 7:
        raise ValueError("weights_option must be between 1 and 7.")

    print("Forecasting properties of the new graph")
    # predict the number of nodes for the predicted graph
    new_nodes, new_nodes_lower, new_nodes_upper = functions.predict_num_nodes(
        graph_list, h=h, conf_level=conf_nodes
    )
    # predict the degrees for the new nodes
    new_nodes_degrees = functions.predict_new_nodes_degrees(graph_list)

    # predict the degrees of the existing nodes
    existing_nodes_degrees = functions.predict_existing_nodes_degrees(
        graph_list, h=h, conf_level=conf_degree
    )

    total_edges = functions.predict_total_edges_counts(
        graph_list, h=h, conf_level=conf_degree
    )

    EDGES_CONF_LEVEL = "conf_high"  # "mean"
    print("Using forecasts to predict the new graph")
    mean_graph = functions.predict_graph_from_forecasts(
        graph_list,
        new_nodes,
        existing_nodes_degrees[EDGES_CONF_LEVEL],
        new_nodes_degrees,
        total_edges[EDGES_CONF_LEVEL],
        conf_nodes,
        conf_degree,
        weights_option,
        weights_param,
    )

    print("Prediction Completed")
    return mean_graph
```

2.4 `read_graph_list(filepath='', sort=True, graph_type='*.gml')`

Reads a list graphs from disk.

The graphs are read in index order, therefore the first graph in the time-series is at index 0.

Parameters:

Name	Type	Description	Default
`filepath`	`str`	The relative path from the root directory to the graphs directory as a string. E.g. "graph_files"	`''`
`sort`	`bool`	A boolean for sorting the graphs contained in the graphs directory alphanumerically. 1 to sort. 0 for OFF.	`True`
`graph_type`	`str`	A string using wildcards to denote the file type of the graphs to be loaded. Must be iGraph compatible e.g. "*.gml*" *The* ".gml" means all gml files found in the filepath are loaded.	`'*.gml'`

Returns:

Type	Description
`list[Graph]`	A list of read graphs.

Source code in `src/netseer/read_graphs.py`

```
7   def read_graph_list(
8       filepath: str = "", sort: bool = True, graph_type: str = "*.gml"
9   ) -> list[ig.Graph]:
10      """Reads a list graphs from disk.
11
12      The graphs are read in index order, therefore the first graph in the time-series is at index 0.
13
14      Args:
15          filepath: The relative path from the root directory to the graphs directory as a string. E.g. "graph_files"
16          sort: A boolean for sorting the graphs contained in the graphs directory alphanumerically. 1 to sort. 0 for OFF.
17          graph_type: A string using wildcards to denote the file type of the graphs to be loaded. Must be iGraph compatible e.g. "*.gml"
18              The "*.gml" means all gml files found in the filepath are loaded.
19
20      Returns:
21          A list of read graphs.
22      """
23      filenames = list()
24      directory_path = Path.cwd() / Path(filepath)
25      # Use glob to get a list of absolute paths for only .gml files.
26      filenames = list(directory_path.glob(graph_type))
27      # Optionally you may need to sort the graph names.
28      if sort:
29          filenames = natsort.natsorted(filenames)
30
31      # read a list of saved graph files
32      print(f"Reading in graph files: {filenames}")
33      return [ig.read(filename) for filename in filenames]
```

2.5 `read_pickled_list(filepath)`

Reads a saved list of graphs from a .pkl file.

Parameters:

Name	Type	Description	Default
`filepath`	`str`	The relative path from the root directory to the pickle file. e.g. "pickled_graphs"	*required*

Returns: A list of graphs.

Source code in `src/netseer/read_graphs.py`

```python
def read_pickled_list(filepath: str) -> list[ig.Graph]:
    """Reads a saved list of graphs from a .pkl file.

    Args:
        filepath: The relative path from the root directory to the pickle file. e.g. "pickled_graphs"
    Returns:
        A list of graphs.
    """
    filename = Path.cwd() / Path(filepath)

    # read a graph list that is stored in a single pickled file
    print(f"Reading pickled graph list from {filename}")
    try:
        with open(filename, "rb") as file:
            data = pickle.load(file)
            return data
    except FileNotFoundError:
        print(f"Error: {filename} not found.")
        quit()
    except Exception as e:
        print(f" An error occurred. {e}")
        quit()
```

Internal: Supplimentary functions for predictions.

Returns a tuple containing the Vertex and Edge error rates, comparing an Actual/Ground Truth Graph to a Predicted Graph.

Parameters:

Name	Type	Description	Default
`actual`	`iGraph`	An Actual Graph to be compared to a Predicted Graph.	*required*
`predicted`	`iGraph`	A Predicted Graph generated by prediction.predict_graph().	*required*

Source code in `src/netseer/measure_error.py`

```python
def measure_error(actual, predicted) -> Tuple[float, float]:
    """Returns a tuple containing the Vertex and Edge error rates, comparing an Actual/Ground Truth Graph to a Predicted Graph.

    Args:
        actual (iGraph): An Actual Graph to be compared to a Predicted Graph.
        predicted (iGraph): A Predicted Graph generated by prediction.predict_graph().

    """
    v_error = vertex_error(actual, predicted)

    e_error = edge_error(actual, predicted)

    return v_error, e_error
```