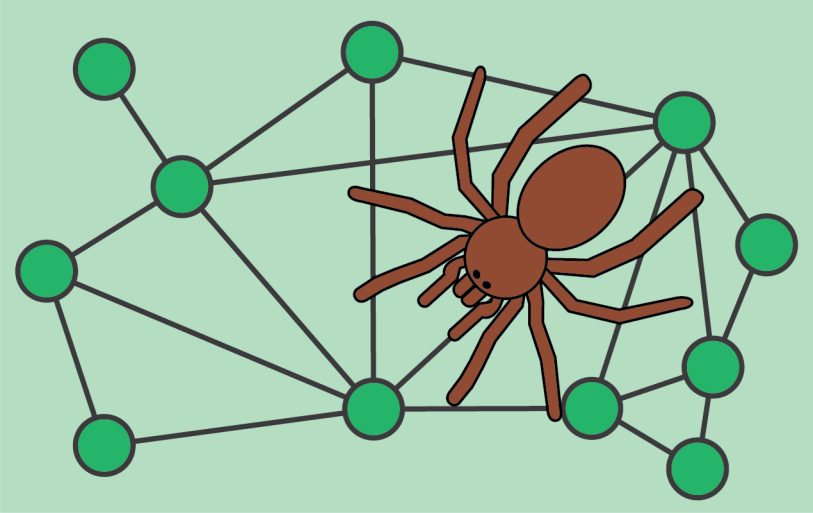
MossSpider Documentation

Release 0.0.3

Paul Zivich

Jul 13, 2022

CONTENTS



`mossspider` provides an implementation of a targeted maximum likelihood estimator (TMLE) for network-dependent data with stochastic policies (network-TMLE) in Python 3.6+. For in-depth details on network-TMLE, see van der Laan (2014), Sofrygin and van der Laan (2017), or Ogburn et al. (2017). `mossspider` get its name from the spruce-fir moss spider, a tarantula that is both the world's smallest tarantula and native to North Carolina.

Network-TMLE is an estimator for causal effects with network-dependent data. Network-TMLE here further relies on a weak dependence assumption (only the immediate contacts of a unit have an effect on that unit's outcome) to make progress in this setting. This is further accomplished via parametric summary measures of immediate contacts' covariates. The following is a brief overview. For further details, please see the references below.

Here, the estimand is the expected mean of an outcome under a policy (indicated by ω) for a super-population of networks each consisting of n units. Due to assumptions for the variance, `mossspider` focuses on this estimand, but is further conditional on the distribution of $\mathbf{W}$ covariates in the network. This estimand can be written as

For identification of ψ, we rely on the following assumptions: causal consistency, exchangeability, and positivity. Respectively, these are written as

$$\text{If } A_i = a, A_i^s = a^s \text{ then } Y_i = Y_i(a, a^s)$$
$$Y(a, a^s) \amalg A, A^s | W, W^s \text{ for all } a \in \mathcal{A}, a^s \in \mathcal{A}^s$$
$$\text{If } \overset{*}{\Pr}(A = a, A^s = a^s | W, W^s) > 0 \text{ then } \Pr(A = a, A^s = a^s | W, W^s) > 0 \text{ for all } a, a^s$$

These assumptions further require that (1) the network is perfectly measured, and (2) the parametric from of the summary measure A^s is known. This set of assumptions is unverifiable and thus needs to be based on substantive knowledge.

Given these assumptions, ψ is identified and we can estimate using network-TMLE. For how network-TMLE operates (as implemented in `mossspider`) see the Overview page on the sidebar.

CHAPTER

ONE

INSTALLATION:

`mossspider` can be downloaded using PyPI. To install `mossspider`, use the following command in terminal or command prompt

```
python -m pip install mossspider
```

There are several dependencies for `mossspider`. These consist of NumPu, SciPy, Pandas, NetworkX, statsmodels, patsy, and Matplotlib. *Note that NetworkX must be at least version 2.0.0 to operate properly*.

To replicate the tests in `tests/` you will need to install `pytest` (but this is not necessary for general use of the package).

CHAPTER

TWO

CONTENTS:

2.1 Overview

Here, we will provide an example application of `mossspider` to highlight some of the available features.

2.1.1 Data Generation

Before demonstrating the application of network-TMLE, we use `mossspider` to generate some generic example data. Here, we will generate both a network and the covariates for that network.

Network

`mossspider` provides a few functions to randomly generate networks with different structural features. Here, we will use the `mossspider.dgm.uniform_network` function. We will generate a network with a uniform degree distribution with degrees between 1-4 and consists of 500 nodes.

```
# importing uniform network
from mossspider.dgm import uniform_network

G = uniform_network(n=500,          # Number of nodes
                    degree=[1, 4],  # Min and Max degree
                    seed=2022)      # Seed for consistency
```

The network generation functions further assign baseline covariates `W` in the network. For the estimand described, `W` is assumed to be held constant in the super-population of networks. Therefore, the data generation step only assigns baseline covariates once. Here, `W` consists of a single binary covariate.

Truth or Reference Values

Next, we will generate data. First, we can use the `mossspider.dgm.generate_truth` function to estimate the mean under the policy of interest, ω. This function takes the specified policy, applies it to the network, calculates the outcomes from the true outcome model, and then returns the mean. To estimate the truth for the super-population of networks, we run this function a 'large' number of times and take the mean of the means. Below is code that does this for a policy where everyone has their probability of action A set to 0.65.

```
import numpy as np
from mossspider.dgm import generate_truth

```

(continues on next page)

(continued from previous page)

```
# Setup values to evaluate at
omega = 0.65                            # Policy of interest
true_p = []                             # Empty storage

# Calculate truth or reference values
for i in range(5000):                   # Sim 5k times
    y_mean = generate_truth(graph=G,    # Mean for graph
                            p=omega)    # ... under omega
    true_p.append(y_mean)               # Store mean

truth = np.mean(true_p)                 # Calculate mean of means
print(truth)
```

Therefore, we have simulated what the estimand is expected to be. Remember, that this estimand will change based on the distribution of $\mathbf{W}$. Therefore, changing the seed in the generation of `G` will result in a different truth value here.

Observed Data

Next, we can simulate the observed data. Instead of using the policy of interest, A and Y are assigned according to some mechanism that is not the policy of interest. In practice, this mechanism is unknown and consists of the nuisance models that must be estimated to use network-TMLE. We will do this using the `mossspider.dgm.generate_observed` function, which returns a network with assigned actions and outcomes

```
from mossspider.dgm import generate_observed

H = generate_observed(G, seed=202203)
```

Notice that if you examine the network, nodes have three attributes: `W`, `A`, and `Y`. `NetworkTMLE` expects the input data to be formatted in a similar manner (a `networkx.Graph` object with assigned node attributes).

2.1.2 Network-TMLE

`mossspider` implements network-TMLE through the `NetworkTMLE` function. The following details how `NetworkTMLE` operates and broadly what happens behind the scenes.

Initialization

As mentioned, `NetworkTMLE` expects the data to be provided in a particular form. This is to ensure all the calculations and data extractions go smoothly behind the scenes. Most importantly, `NetworkTMLE` expects the data to be provided as a `networkx.Graph` object. Furthermore, all covariates must be provided as node attributes.

Below is the initialization of `NetworkTMLE` for the previously generated data set

```
ntmle = NetworkTMLE(network=H,      # NetworkX graph
                    exposure='A',   # Exposure in graph
                    outcome='Y',    # Outcome in graph
                    verbose=True)   # Print model summaries
```

Besides the network, `NetworkTMLE` requires that the label for the action (referred to as exposure here) nad the label for the outcome in the graph are provided. There are optional arguments for the confidence-level (`alpha`), whether to

apply a restriction based on degree (`degree_restrict`), and whether to display nuisance model summary information (`verbose`). By default no degree restriction is applied and 95% confidence intervals are provided.

Behind the scenes, `NetworkTMLE` extracts the covariates from the graph, creates a `pandas.DataFrame`, and calculates summary measures. Covariates are provided via `networkx.Graph` instead of a `pandas.DataFrame` to ensure that summary measures are all correctly calculated. This is then merged with the degree of each node (with the optional degree restriction applied). Finally, storage for intermediate pieces are created. Continuous outcomes are further bounded to be $(0, 1)$ for the targeting step later on.

Exposure Nuisance Model

Next, we need to specify the exposure nuisance model. These models are used to calculate the following weights:

$$\frac{\Pr^*(A, A^s|W, W^s)}{\Pr(A, A^s|W, W^s)}$$

where the numerator is from the policy of interest and the denominator is based on the observed distribution of actions. Here, we estimate these models by factoring the probabilities as

$$\Pr(A, A^s|W, W^s) = \Pr(A|W, W^s) \Pr(A^s|A, W, W^s)$$

Therefore, two models need to be specified: one for A, and one for A^s. For A, we will use a logistic model

```
# Model for Pr(A | W, W^s)
ntmle.exposure_model(model="W + W_sum",  # Parametric model
                     custom_model=None)  # ... optional argument
```

Certain flexible models (e.g, sci-kit learn models) can also be used. Note that these must be classifiers and are provided via the optional `custom_model` argument.

Next, a model for the summary measure needs to be specified. Importantly, the summary measure and an appropriate model must be selected. For available summary measures, see the Summary Measures page. Here, we will use the following summary measure

$$A_i^s = \sum_{j=1}^{n} A_j \mathcal{G}_{ij}$$

where $\mathcal{G}$ is the adjacency matrix. This summary measure is a simple count of the immediate contacts with $A = 1$. Now, we can specify the exposure mapping model

```
# Model for Pr(A^s | A, W, W^s)
ntmle.exposure_map_model(model='A + W + W_sum',  # Parametric model
                         measure='sum',          # Summary measure for A^s
                         distribution='poisson') # Model distribution to use
```

Here, the model must be provided as well as the summary measure (`measure`) and the distribution to use for the model (`distribution`). Since our summary measure is a count, we use a Poisson regression model. While `custom_models` are provided, care must be taken to ensure that the distribution of that custom model agrees with the `distribution` argument. Otherwise, weights **will not** be estimated correctly.

In both of these steps, we are only specifying the parametric form of these models and the summary measures to use. The actual estimation of the weights is done later in the `NetworkTMLE.fit` step.

Outcome Nuisance Model

Next, we need to specify and estimate the outcome nuisance model: $E[Y|A, A^s, W, W^s]$. Unlike the weights, we can (and will) estimate the outcome model in this function. To specify the outcome model

```
# Model for E[Y | A, A^s, W, W^s]
ntmle.outcome_model(model='A + A_sum + W + W_sum',
                    custom_model=None)
```

For binary outcomes (internally detected in the initialization), a logistic model is used. For continuous outcomes, the default is linear regression but other models can be used by specifying the optional `distribution` argument. Finally, custom models can also be used here. There is more flexibility in what algorithms could be considered (since we only need the predicted values).

Notice that the summary measure for the outcome nuisance model and the exposure nuisance model are the same for A^s.

Behind the scenes, the function saves the model specification, fits the specified outcome model, and generates predicted values of the outcome under the observed values of A and A^s. These estimates are all stored interally for the next step.

Estimation

Finally, we can estimate the conditional mean under the policy of interest. `NetworkTMLE` takes the policy in the form of a float (which sets everyone to the same probability of having $A = 1$) or as a vector (assigns each unit their own probability of $A = 1$). Here, the policy of interest is $\Pr(A_i) = 0.65$.

```
# Estimation
ntmle.fit(p=0.65,           # Policy
          samples=500,      # ... replicates for MC integration
          bound=None,       # ... option to bound weights
          seed=20220316)    # ... seed for consistency
```

Other optional arguments include settings the number of samples to use in the Monte Carlo integration procedure (`samples`, see below for details on this), truncation of estimated weights (`bound`), and a random seed for consistent results of the estimation procedure.

Behind the scenes, there are lots of steps that occur. First, checks are applied to make sure the nuisance models are all specified and the policy is been specified in a compatible format. Next, the weights are estimated. This is done by estimating the denominator using the observed data. For the numerator, we can't use the policy of interest directly (since it is specified in terms of A_i and not A_i, A_i^s). Therefore, we use a Monte-Carlo procedure. Briefly, we generate `samples` copy of the data. To each copy, the stochastic policy is applied. Using all copies of the data with the copy of the stochastic policy applied simultaneously, the exposure nuisance models are estimated. Then the observed A_i, A_i^s and estimated model parameters are used to estimate the numerator. If `bound` is specified, the weights are then bounded.

Next, the targeting step is applied. This involves taking the predicted values from `NetworkTMLE.outcome_model` and the estimated weights and fitting a weighted intercept-only logistic model. Then the outcome model is used to predict the outcome under the policy of interest and is updated using the estimated targeting model. Since stochastic policies have a number of different possible distributions, a Monte-Carlo procedure is again used. Here, we re-use the data sets generated in the weight estimation step. Using the A_i, A_i^s under the policy, predicted values of the outcomes are generated, updated via the targeting model, averaged over each data set, and finally averaged across the `samples`.

Finally, the variance is calculated. Two variances are calculated. The first assumes that all dependence is due to direct transmission only, while the second allows for direct and latent transmission. For theoretical reasons, the latter will generally be preferred.

Note that increasing `samples` will result in a more 'stable' estimate (it will be less subject to random noise if a different seed had been used). Personally, I have found good performance with 100-500. Ideally, you would run as much as

possible. Unfortunately, the most computationally intensive part is the generation of copies of the data set. Therefore, run-times are highly dependent on the value used for `samples`.

Summary Results

A summary of the results can be printed to the console via:

```
# Displaying results
ntmle.summary(decimal=4)
```

To increase the number of decimals displayed, use the `decimal` argument.

Diagnostic

Finally, we have a diagnostic available. The diagnostic provides a plot to visually assess how well-supported the policy of interest is by the observed distribution of $\mathbf{A}$. Briefly, the diagnostic plots the summary measure A_i^s by A_i in the observed data. This is then contrasted with A_i^s under the policy (as generated in the Monte-Carlo step). For well-supported policies, the observed data and generated data under the policy should overlap. If there is little overlap, this is indicative of the policy of interest being poorly-supported by the data. Poorly-supported policies can result in biased estimation and poor confidence interval coverage. For details see [...].

The diagnostic plot can be generated via

```
import matplotlib.pyplot as plt

ntmle.diagnostics()
plt.show()
```

2.1.3 Additional Examples

Additional examples are provided here.

2.2 Summary Measures

The following provides documentation for the available summary measures in `mossspider`. Currently, `mossspider` does not support fully custom summary measures. We are working on how to best implement this option.

Column names for all summary measures currently in the data can be checked via

```
ntmle = NetworkTMLE(G, exposure='A', outcome='Y')
print(ntmle.df.columns)
```

2.2.1 Basic Measures

The following basic summary measures are always available for model specifications. They are all calculated by default. They include: sum, mean, variance, mean distance, and variance distance.

Throughout this section, let X indicate the covariate the summary measure is being calculated for, and $\mathcal{G}$ indicate the adjacency matrix for the network.

Sum

The sum summary measure is defined as

$$X_i^s = \sum_{i=1}^{n} X_j \mathcal{G}_{ij}$$

For the covariate `X` in the data, the sum summary measure column is accessed by `X_sum`.

Mean

The mean summary measure is defined as

$$X_i^s = \frac{\sum_{i=1}^{n} X_j \mathcal{G}_{ij}}{\sum_{i=1}^{n} \mathcal{G}_{ij}}$$

For the covariate `X` in the data, the mean summary measure column is accessed by `X_mean`.

Variance

The variance summary measure is defined as

$$X_i^s = \frac{\sum_{i=1}^{n} (X_j - X_i)^2 \mathcal{G}_{ij}}{\sum_{i=1}^{n} \mathcal{G}_{ij}}$$

For the covariate `X` in the data, the variance summary measure column is accessed by `X_var`.

Mean Distance

The mean distance summary measure is defined as

$$X_i^s = \frac{\sum_{i=1}^{n} (X_j - X_i) \mathcal{G}_{ij}}{\sum_{i=1}^{n} \mathcal{G}_{ij}}$$

For the covariate `X` in the data, the mean distance summary measure column is accessed by `X_mean_dist`.

Variance Distance

The variance distiance summary measure is defined as

$$X_i^s = \frac{\sum_{i=1}^{n} ((X_j - X_i) - \bar{X}_i)^2 \mathcal{G}_{ij}}{\sum_{i=1}^{n} \mathcal{G}_{ij}}$$

where

$$\bar{X}_i = \frac{\sum_{i=1}^{n} X_j \mathcal{G}_{ij}}{\sum_{i=1}^{n} \mathcal{G}_{ij}}$$

For the covariate `X` in the data, the variance distance summary measure column is accessed by `X_var_dist`.

2.2.2 Partially Custom Measures

Partially custom measures allow for some flexibility. These measures include a threshold measure and a category measure. By default, these are not automatically calculated. They must be specified using the corresponding functions.

These measures are further build upon the basic measures. Therefore, familiarize yourself with the basic measures before this section.

Threshold

For a specified variable and summary measure, a threshold indicator variable can be created. For example, we may want to create a summary measure of the action which is an indicator if a unit has more than 3 immediate contacts with $A = 1$. To create this measure, we call

```
ntmle.define_threshold(variable='A_sum',    # Variable to use
                       threshold=3)         # ... set threshold (>, or <=)
```

This function should be called prior to estimating the nuisance models. Furthermore, the function calculates the threshold measure for the observed data and the Monte-Carlo generated data automatically.

Thresholds can be created for multiple variables by specifying the `define_threshold` argument multiple times.

To access the threshold summary measure column, use `'A_sum_t3'` here. The naming convention works like the following: variable + underscore + t + threshold.

Category

For a specified variable and summary measure, a categorization is created. For example, we may want to bin `W_sum` to reduce the dimension for a power-law network while still trying to model `W_sum` flexibly. To create a category summary measure based on user-specified bins, the following function is used:

```
ntmle.define_category(variable='W_sum',        # Variable to bin
                      bins=[0, 1, 3, 7, 12],   # ... bins (includes right)
                      labels=False)            # ... allow for new labels (not␣
↪recommended)
```

From this function, a new column consisting of a categorical dummy variable is generated. The naming convention for this new column is the variable name + underscore + c. Therefore, the new categorical variable would be `'W_sum_c'`.

As with the threshold, this function should be called prior to estimating the nuisance models. Furthermore, the function calculates the threshold measure for the observed data and the Monte-Carlo generated data automatically. Finally, categories can be created for multiple variables by specifying the `define_category` argument multiple times.

2.2.3 Fully Custom Measures

Not available yet.

2.3 Reference

Documentation for available functions and arguments for those functions are provided here.

2.3.1 Estimators

NetworkTMLE(network, exposure, outcome[, ...])	Implementation of the Targeted Maximum Likelihood Estimator (TMLE) for network dependent data.

mossspider.estimators.tmle.NetworkTMLE

class NetworkTMLE(*network*, *exposure*, *outcome*, *degree_restrict=None*, *alpha=0.05*, *continuous_bound=0.0005*, *verbose=False*)

Implementation of the Targeted Maximum Likelihood Estimator (TMLE) for network dependent data. The following procedure estimates the expected incidence under a treatment plan of interest. For stochastic treatment plans, the expected incidence is obtained through Monte Carlo integration of a subsample of possible treatment allotments that correspond to the plan of interest.

Note: Network-TMLE makes the weak dependence assumption, such that only direct contacts' treatment can interfere with individual i's outcome.

Parameters

- **network** (*NetworkX Graph*) – NetworkX undirected network *without* self-loops. Additionally, all variables should be stored as attributes for each node. `Targetula` extracts the node data from the graph and creates a `pandas.DataFrame` object from that information. It is important that no nodes have missing data. Currently there is no procedure to handle missing data
- **exposure** (*str*) – String indicating the exposure variable of interest.
- **outcome** (*str*) – String indicating the outcome variable of interest.
- **degree_restrict** (*None, list, tuple, optional*) – Restriction on the minimum & maximum degree for nodes to be included in the estimand. Must be a list with a length of two, where the first value corresponds to the lower bound and the second is the upper bound for degree. Values are inclusive. All samples below the first value OR above the second level are considered as "background" features. Hence the intervention does not change their exposure.
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05
- **continuous_bound** (*float, optional*) – For continuous outcomes, TMLE needs to bound Y between 0 and 1. However, 0/1 cannot be included in these bounded values. This specification sets the bounds for the continuous outcomes. The default is 0.0005.
- **verbose** (*bool, optional*) – Whether to print all intermediary model results for the estimation process. When set to True, each of the model results are printed to the console. The default is False.

Note: `mossspider` calculates exposure mapping variables automatically with the input network. These vari-

ables are saved as variable-name_map. So for a variable *'A'*, the newly created exposure mapping variable calculated is *'A_map'*

Note: For directed networks, the direction of of influence goes from the target node to the source (i.e. opposite of the arrow direction). If *A –> B* then B's covariates will be part of the A's summary measures.

Examples

Setting up environment

```
>>> from mossspider import NetworkTMLE
>>> from mossspider.dgm import uniform_network, generate_observed
```

Generating a generic network and some data

```
>>> graph = generate_observed(uniform_network(n=500, degree=[1, 6]))
```

Estimation with *NetworkTMLE* (nonparametric summary measure in exposure map model)

```
>>> tmle = NetworkTMLE(network=graph, exposure='A', outcome='Y')
>>> tmle.exposure_model('W + W_map')
>>> tmle.exposure_map_model('A + W + W_map', distribution=None)
>>> tmle.outcome_model('A + W + A_map + W_map', print_results=False)
>>> tmle.fit(p=0.8, bound=10e5)
>>> tmle.summary()
```

Estimation with *NetworkTMLE* (parametric summary measure in exposure map model)

```
>>> tmle = NetworkTMLE(network=graph, exposure='A', outcome='Y')
>>> tmle.exposure_model('W + W_map')
>>> tmle.exposure_map_model('A + W + W_map', measure='sum', distribution='poisson')
>>> tmle.outcome_model('A + W + A_map + W_map', print_results=False)
>>> tmle.fit(p=0.8, bound=10e5)
>>> tmle.summary()
```

Estimation with *NetworkTMLE* and restricting inference by degree

```
>>> tmle = NetworkTMLE(network=graph, exposure='A', outcome='Y', degree_restrict=[0,
↪ 5])
>>> tmle.exposure_model('W + W_map')
>>> tmle.exposure_map_model('A + W + W_map', measure='sum', distribution='poisson')
>>> tmle.outcome_model('A + W + A_map + W_map', print_results=False)
>>> tmle.fit(p=0.8, bound=10e5)
>>> tmle.summary()
```

Diagnostic plot for support of policy of interest in observed data

```
>>> import matplotlib.pyplot as plt
>>> tmle.diagnostics()
>>> plt.show()
```

Generating a threshold measure based on a summary measure

```
>>> tmle = NetworkTMLE(network=graph, exposure='A', outcome='Y')
>>> tmle.define_threshold(variable='A_sum', threshold=3)  # A_sum_t3
```

Generating a category measure based on a binned summary measure

```
>>> tmle = NetworkTMLE(network=graph, exposure='A', outcome='Y')
>>> tmle.define_category(variable='A_sum', bins=[0, 1, 2, 4, 6])  # A_sum_c
```

References

van der Laan MJ. (2014). Causal inference for a population of causally connected units. *Journal of Causal Inference*, 2(1), 13-74.

Sofrygin O & van der Laan MJ. (2017). Semi-parametric estimation and inference for the mean outcome of the single time-point intervention in a causally connected population. *Journal of Causal Inference*, 5(1).

Ogburn EL, Sofrygin O, Diaz I, & van der Laan MJ. (2017). Causal inference for social network data. *arXiv preprint* arXiv:1705.08527.

Sofrygin O, Ogburn EL, & van der Laan MJ. (2018). Single Time Point Interventions in Network-Dependent Data. In Targeted Learning in Data Science (pp. 373-396). Springer.

__init__(*network*, *exposure*, *outcome*, *degree_restrict=None*, *alpha=0.05*, *continuous_bound=0.0005*, *verbose=False*)

Methods

__init__(network, exposure, outcome[, ...])	

define_category(variable, bins[, labels])	Function arbitrarily allows for multiple different defined thresholds
define_threshold(variable, threshold)	Function arbitrarily allows for multiple different defined thresholds
diagnostics([figsize, color_a1, color_a0])	Returns diagnostic plot for the specified network-TMLE.
exposure_map_model(model[, measure, ...])	Exposure summary measure model for individual i.
exposure_model(model[, custom_model, ...])	Exposure model for individual i.
fit(p[, samples, bound, seed])	Estimation procedure under a specified treatment plan.
outcome_model(model[, custom_model, ...])	Estimation of the outcome model E(Y|A, A_map, W, W_map).
summary([decimal])	Prints summary results for the sample average treatment effect under the treatment plan specified in the fit procedure

exposure_model(*model*, *custom_model=None*, *custom_model_sim=None*)

Exposure model for individual i. Estimates Pr(A=a|W, W_map) using a logistic regression model.

> **Note:** This function only saves the model specifications. IPTW are calculated later during the fit() procedure since the policy is needed.

Parameters

- **model** (*str*) – Exposure mapping model. Ideally would include treatment for individual i
- **custom_model** – User-specified model
- **custom_model_sim** – User-specified model. This allows the user to specify a different IPW model to be fit for the numerator. That model is fit to the simulated data, so some constraints may be added to speed up the estimation procedure. If None and custom_model is not None, copies over the custom_model used.

exposure_map_model(*model*, *measure=None*, *distribution=None*, *custom_model=None*, *custom_model_sim=None*)

Exposure summary measure model for individual i. Estimates Pr(A_map=a|A=a, W, W_map) using a logistic regression model.

Note: Only saves the model specifications. IPTW are calculated later during the fit() function

There are several options for the distributions of the summary measure. One option is a non-parametric approach that estimates the probability for each individual contact (works best for uniform distributions). However, this approach may not always be possible to estimate. Instead, parametric distributional assumption can be used instead. Currently, implemented are normal and Poisson distributions.

Parameters

- **model** (*str*) – Exposure mapping model. Ideally would include treatment for individual i
- **measure** (*None, str, optional*) – Exposure mapping to use for the modeling statement. Options include 'mean' and 'sum'. Default is None which natively works with the *distribution=None* option
- **distribution** (*None, str, optional*) – Distribution to use for exposure mapping model. Options include: non-parametric (None), Normal ('normal'), Poisson ('poisson').
- **custom_model** (*None, optional*) – User-specified model
- **custom_model_sim** – User-specified model. This allows the user to specify a different IPW model to be fit for the numerator. That model is fit to the simulated data, so some constraints may be added to speed up the estimation procedure. If None and custom_model is not None, copies over the custom_model used.

outcome_model(*model*, *custom_model=None*, *distribution='normal'*)

Estimation of the outcome model E(Y|A, A_map, W, W_map).

Note: Estimates the outcome model (g-formula) using the observed data and generates predictions under the observed distribution of the exposure.

Parameters

- **model** (*str*) – Specified Q-model
- **custom_model** – User-specified model
- **distribution** (*optional, str*) – For non-binary outcome variables, the distribution of Y must be specified. Default is 'normal'.

fit(*p*, *samples=100*, *bound=None*, *seed=None*)

Estimation procedure under a specified treatment plan.

This function estimates the IPTW for the treatment plan of interest, performs the target steps, and performs Monte Carlo integration with the targeted model, and calculates confidence intervals. Confidence intervals are obtained from influence curves.

Parameters

- **p** (*float, int, list, set*) – Percent of population to treat. For conditional treatment plans, a container object of floats. All values must be between 0 and 1
- **samples** (*int*) – Number of samples to generate to calculate numerator for weights and for the Monte Carlo integration procedure for stochastic treatment plans. For deterministic treatment plans (p==1 or p==0), samples is set to 1 to reduce computation burden. Deterministic treatment plan do not require the Monte Carlo integration procedure
- **bound** (*None, int, float*) – Bounds to truncate calculate weights by...
- **seed** (*int, None*) – Random seed for the Monte Carlo integration procedure

summary(*decimal=3*)

Prints summary results for the sample average treatment effect under the treatment plan specified in the fit procedure

Parameters **decimal** (*int*) – Number of decimal places to display

Returns

Return type None

diagnostics(*figsize=(6, 5)*, *color_a1='blue'*, *color_a0='red'*)

Returns diagnostic plot for the specified network-TMLE. The currently available diagnostic presents plots of the designated summary measure for A^s (stratified by A) for the observed data, and the Monte Carlo simulated data. This diagnostic can be used to visually assess whether the designated policy is poorly-supported by the data.

Note: A policy that has little overlap with the observed data is indicative of the policy being poorly supported by the observed data. Poorly-supported policies may not be well estimated and thus considering other stochastic policies in recommended.

Parameters

- **figsize** (*list, set, array, optional*) – Determine the figure size (dimensions). Passes directly to `plt.subplots(...figsize=figsize)`.
- **color_a1** (*str, optional*) – Color for the A=1 group in the figure. Default is blue.
- **color_a0** (*str, optional*) – Color for the A=0 group in the figure. Default is red.

Returns

Return type Diagnostic plot for data support of policy.

define_threshold(*variable*, *threshold*)

Function arbitrarily allows for multiple different defined thresholds

Parameters

- **variable** (*str*) – Variable to generate categories for
- **threshold** (*int, float*) – Threshold to use as the cutpoint.

define_category(*variable*, *bins*, *labels=False*)
Function arbitrarily allows for multiple different defined thresholds

Parameters

- **variable** (*str*) – Variable to generate categories for
- **bins** (*list, set, array*) – Bin cutpoints to generate the categorical variable for. Uses `pandas.cut(..., include_lowest=True)` to create the binned variables.
- **labels** (*list, set, array*) – Specified labels. Can be given custom labels, but generally recommend to keep set as False

2.3.2 Data Generation

uniform_network(n, degree[, pr_w, seed])	Generates a uniform random graph for a set number of nodes (n) and specified max and min degree (degree).
clustered_power_law_network(n_cluster[, ...])	Generate a graph with the following features: follows a power-law degree distribution, high(er) clustering coefficient, and an underlying community structure.
generate_observed(graph[, seed])	Simulates the exposure and outcome for the uniform random graph (following mechanisms are from Sofrygin & van der Laan 2017).
generate_truth(graph, p)	Simulates the true conditional mean outcome for a given network, distribution of W, and policy.

mossspider.dgm.uniform_network

uniform_network(*n*, *degree*, *pr_w=0.35*, *seed=None*)
Generates a uniform random graph for a set number of nodes (n) and specified max and min degree (degree). Additionally, assigns a binary baseline covariate, W, to each observation.

Parameters

- **n** (*int*) – Number of nodes in the generated network
- **degree** (*list, set, array*) – An array of two elements. The first element is the minimum degree and the second element is the maximum degree.
- **pr_w** (*float, optional*) – Probability of W=1. W is a binary baseline covariate assigned to each unit.
- **seed** (*int, None, optional*) – Random seed to use. Default is None.

Returns

Return type networkx.Graph

Examples

Loading the necessary functions

```
>>> from mossspider.dgm import uniform_network
```

Generating the uniform network

```
>>> G = uniform_network(n=500, degree=[0, 2])
```

mossspider.dgm.clustered_power_law_network

clustered_power_law_network(*n_cluster*, *edges=3*, *pr_cluster=0.75*, *pr_between=0.0007*, *pr_w=0.35*, *seed=None*)

Generate a graph with the following features: follows a power-law degree distribution, high(er) clustering coefficient, and an underlying community structure. This graph is created by generating a number of subgraphs with power-law distributions and clustering. The subgraphs are generated using `networkx.powerlaw_cluster_graph(n=n_cluster[...], m=edges, p=p_cluster)`. This process is repeated for each element in the `n_cluster` argument. Then the subgraphs are then randomly connected by creating random edges between nodes of the subgraphs.

Parameters

- **n_cluster** (*list, set, array, ndarray*) – Specify the N for each subgraph in the clustered power-law network via a list. List should be positive integers that correspond to the N for each subgraph.
- **edges** (*int, optional*) – Number of edges to generate within each cluster. Equivalent to the `m` argument in `networkx.powerlaw_cluster_graph`.
- **pr_cluster** (*float, optional*) – Probability of a new node forming a triad with neighbors of connected nodes
- **pr_between** (*float, optional*) – Probability of an edge between nodes of each cluster. Evaluated for all node pairs, so should be relatively low to keep a high community structure. Default is 0.0007.
- **pr_w** (*float, optional*) – Probability of the binary baseline covariate W for the network. Default is 0.35.
- **seed** (*int, None, optional*) – Random seed. Default is None.

Returns

Return type networkx.Graph

Examples

Loading the necessary functions

```
>>> from mossspider.dgm import clustered_power_law_network
```

Generating the clustered power-law network

```
>>> G = clustered_power_law_network(n_cluster=[50, 50, 50, 50])
```

mossspider.dgm.generate_observed

generate_observed(*graph*, *seed=None*)

Simulates the exposure and outcome for the uniform random graph (following mechanisms are from Sofrygin & van der Laan 2017).

$$A = \text{Bernoulli}(\text{expit}(-1.2 + 1.5W + 0.6W^s))$$
$$Y = \text{Bernoulli}(\text{expit}(-2.5 + 0.5A + 1.5A^s + 1.5W + 1.5W^s))$$

Parameters

- **graph** (*Graph*) – Graph generated by the *uniform_network* function.
- **seed** (*int, None, optional*) – Random seed to use. Default is None.

Returns

Return type Network object with node attributes

Examples

Loading the necessary functions

```
>>> from mossspider.dgm import uniform_network, generate_observed
```

Generating the uniform network

```
>>> G = uniform_network(n=500, degree=[0, 2])
```

Generating exposure A and outcome Y for network

```
>>> H = generate_observed(graph=G)
```

References

Sofrygin O, & van der Laan MJ. (2017). Semi-parametric estimation and inference for the mean outcome of the single time-point intervention in a causally connected population. *Journal of Causal Inference*, 5(1).

mossspider.dgm.generate_truth

generate_truth(*graph*, *p*)

Simulates the true conditional mean outcome for a given network, distribution of W, and policy.

The true mean under the policy is simulated as

$$A = Bernoulli(p)\ Y = Bernoulli(expit(-2.5 + 1.5 * W + 0.5 * A + 1.5 * map(A) + 1.5 * map(W)))$$

Returns

Return type float

Examples

Loading the necessary functions

```
>>> from mossspider.dgm import uniform_network, generate_truth
```

Generating the uniform network

```
>>> G = uniform_network(n=500, degree=[0, 2])
```

Calculating truth for a policy via a large number of replicates

```
>>> true_p = []
>>> for i in range(1000):
>>>     y_mean = generate_truth(graph=G, p=0.5)
>>>     true_p.append(y_mean)
>>> np.mean(true_p)  # 'true' value for the stochastic policy
```

To reduce random error, a large number of replicates should be used

2.3.3 Utilities

Plan to add some basic utilities in a future version (such as helping setup the network as `NetworkTMLE` expects).

CHAPTER

THREE

CODE AND ISSUE TRACKER

Please report bugs, issues, or feature requests on GitHub at pzivich/MossSpider.

Otherwise, you may contact us via email (gmail: zivich.5) or on Twitter (@PausalZ)

CHAPTER

FOUR

REFERENCES

Ogburn EL, Sofrygin O, Diaz I, & van der Laan, MJ. (2017). Causal inference for social network data. *arXiv preprint arXiv:1705.08527*.

Sofrygin O, & van der Laan MJ. (2017). Semi-parametric estimation and inference for the mean outcome of the single time-point intervention in a causally connected population. *Journal of Causal Inference*, 5(1).

van der Laan MJ. (2014). Causal inference for a population of causally connected units. *Journal of Causal Inference*, 2(1), 13-74.

INDEX

Symbols

C

D

E

F

G

N

O

S

U