Network Edges Inference

General Principles

This document implements the BISoN framework methodologies Hart et al. (2023) that enables modeling edge weights with uncertainty by decomposing aggregated observations into the generative sampling processes. Below, we detail the implementation of various edge weight models for different types of observational data using the Bayesian Inference (**BI**) package via Python, R, and Julia.

The fundamental principle behind the Network Edges Inference models is to capture uncertainty in edge weights by modeling the data collection process directly, rather than relying on point estimates (e.g., Simple Ratio Index). By decomposing the data into distinct observation periods, these models can inherently capture the uncertainty generated by sampling effort and allow for observation-level effects (such as variable visibility) to be controlled.

Three main types of observational data are supported: 1. **Binary data**: Presence/absence of social events in fixed sampling periods. 2. **Count data**: Number of social events per sampling period. 3. **Duration data**: Amount of time spent engaged in social events within a sampling period.

Binary Data Model

- This model is a more specific case of Network model, where we assume that the network is binary and that the nodes are exchangeable.

General Principles

For each models types Hart et al. (2023) provided the possibility to account for: 1. **Directionality**: Directional or non-directional edge weights can be modeled. 2. **Fixed effects**: Additional variables accounting for fixed effects (e.g., location or age difference) can be added to the predictor iteratively. 3. **Random effects**: Additional variables accounting for nodal random effects (e.g., location or age difference) can be added to the predictor iteratively. 4.

Partial pooling : Additional variables accounting for dyads random effects (e.g., goup belonging) can be added to the predictor iteratively. 5. **Zero-inflation**: Additional variables accounting for random effects (e.g., location or age difference) can be added to the predictor iteratively.

General Principles

The binary data model is used when the presence or absence of a social event for each dyad is recorded in each sampling period. The edge weight represents the probability of (or proportion of time) engaging in a social event in a fixed period of time.

Considerations

> **i** Note
>
> - **Priors**: The edge weight parameters normally represent logit-scale proportions and typically use standard Normal priors relative to prior structural beliefs (e.g., `Normal(0, 2)`).
> - **Data**: The model uses the number of sampling periods (`divisor`) where an event could occur and the number of observed events (`event`).
> - **Fixed Effects**: Additional variables accounting for fixed effects (e.g., location or age difference) can be added to the predictor iteratively.

Example with simulated data

Simulated data can be generated with **bisonR** function `simulate_bison_model`. For all example we will account the presence of all possible features: 1 fixed nodal predictors, 1 random nodal predictors, presence of subgroups, partial_pooling and zero inflated model. This will allow you to adapt the model by removing parts of the model based on your data.

To be compatible with `data` used in the model `data` must contain and be zero-based indexed (i.e. dyad_idsID, num_random_groups need to start at 0): 1. *num_rows* : Number of observations for the total period of time. 2. *event* : For binary model whether event occur (1) or not (0). 3. *duration* : Duration for each observations. 4. *dyad_ids* : ID for each possible combinations of dyad. 5. *num_edges*: For binary model this is equivalent of *num_rows*. 6. *num_fixed*: Number of fixed nodal predictors. 7. *num_random*: Number of random nodal predictors. 8. *num_random_groups* : Number of subgroups. 9. *random_group_index* : Subgroups ID. 10. *design_fixed*: Design matrix (N, P) for nodal predictors, where N is the number of individuals and P the number of predictors. 11. *design_random*: Design matrix for nodal random effects.

12. *partial_pooling*: Indicator (0 or 1) for enabling/disabling partial pooling of edge weights.
13. *zero_inflated*: Indicator (0 or 1) for enabling/disabling zero-inflation.

Python

```python
from BI import bi

m = bi(platform='cpu')

m.data_on_model = {'data': data_list_jax} # Must contain 'prior_edge_mu', 'num_edges', 'dyad

def bi_model_binary(data):
    # Edge weights prior
    edge_weight = m.dist.normal(data['prior_edge_mu'], data['prior_edge_sigma'],
                                shape=(data['num_edges'],), name="edge_weight")

    # 0-based indexing for JAX
    dyad_ids_0 = data['dyad_ids'] - 1

    # Build predictor
    predictor = edge_weight[dyad_ids_0]

    # Add fixed effects if applicable
    if data['num_fixed'] > 0:
        beta_fixed = m.dist.normal(data['prior_fixed_mu'], data['prior_fixed_sigma'],
                                   shape=(data['num_fixed'],), name="beta_fixed")
        predictor = predictor + m.jnp.dot(data['design_fixed'], beta_fixed)

    # Convert from logit scale to probability
    p = m.jax.scipy.special.expit(predictor)

    # Binomial Likelihood
    m.dist.binomial(data['divisor'], p, obs=data['event'], name="event")

m.fit(bi_model_binary)
m.summary()
```

R

```r
library(BayesianInference)
m <- importBI(platform = "cpu")
jnp <- reticulate::import("jax.numpy")
jax_scipy <- reticulate::import("jax.scipy.special")

m$data_on_model <- list(data = data_list_jax)

bi_model_binary <- function(data) {
  # Edge weights prior
  edge_weight <- m$dist$normal(data$prior_edge_mu, data$prior_edge_sigma,
                               shape=tuple(data$num_edges), name="edge_weight")

  # 0-based indexing for JAX
  dyad_ids_0 <- data$dyad_ids - 1L

  # Build predictor
  predictor <- edge_weight[dyad_ids_0]

  if (data$num_fixed > 0) {
    beta_fixed <- m$dist$normal(data$prior_fixed_mu, data$prior_fixed_sigma,
                                shape=tuple(data$num_fixed), name="beta_fixed")
    predictor <- predictor + jnp$dot(data$design_fixed, beta_fixed)
  }

  p <- jax_scipy$expit(predictor)

  m$dist$binomial(data$divisor, p, obs=data$event, name="event")
}

m$fit(bi_model_binary)
m$summary()
```

Julia

```julia
using BayesianInference

m = importBI(platform="cpu")
m.data_on_model = Dict("data" => data_list_jax)

@BI function bi_model_binary(data)
```

```
    edge_weight = m.dist.normal(data["prior_edge_mu"], data["prior_edge_sigma"],
                                shape=(data["num_edges"],), name="edge_weight")

    # 0-based indexing for JAX inside the environment
    dyad_ids_0 = data["dyad_ids"] .- 1

    predictor = edge_weight[dyad_ids_0]

    if data["num_fixed"] > 0
        beta_fixed = m.dist.normal(data["prior_fixed_mu"], data["prior_fixed_sigma"],
                                   shape=(data["num_fixed"],), name="beta_fixed")
        predictor = predictor .+ jnp.dot(data["design_fixed"], beta_fixed)
    end

    p = jax.scipy.special.expit(predictor)

    m.dist.binomial(data["divisor"], p, obs=data["event"], name="event")
end

m.fit(bi_model_binary)
m.summary()
```

Mathematical Details

For binary data, the mathematical process applies a Binomial likelihood assuming multiple Bernoulli trials:

$$X_{ij}^{(n)} \sim \text{Binomial}(D_{ij}^{(n)}, p_{ij}^{(n)})$$

Where $X_{ij}^{(n)}$ is the presence/absence in the n-th observation, $D_{ij}^{(n)}$ is the condition (often 1 or representing multiple grouped trials), and $p_{ij}^{(n)}$ is the dyad interaction probability. The model logit-links the predictor effects with the primary edge weight:

$$\text{logit}(p_{ij}^{(n)}) = \omega_{ij} + \ldots$$

Where ω_{ij} represents the foundational edge weight parameter for dyad i, j.

Notes

> **Note**
>
> The binary binomial implementation directly provides a framework mirroring the Simple Ratio Index (SRI) while preserving variance from low sample depth.

Count Data Model

General Principles

The count data model interprets instances where the exact *number of events* for each dyad is recorded in each sampling period. Under this formulation, the model outputs a rate of occurrence of social events per unit of time instead of a bounded probability.

Considerations

> **Note**
>
> - **Priors**: Counts use a standard Poisson process; therefore, edge predictors dictate the log-rate (λ) and must be carefully scaled relative to time. `Normal` priors on the log-rate parameter effectively capture the magnitudes.
> - **Link Function**: Uses the exponential function applied to the linear predictor configuration.

Example with simulated data

Python

```python
from BI import bi

m = bi(platform='cpu')
m.data_on_model = {'data': data_list_jax}

def bi_model_count(data):
    # Edge weights prior (log-rate)
    edge_weight = m.dist.normal(data['prior_edge_mu'], data['prior_edge_sigma'],
                                shape=(data['num_edges'],), name="edge_weight")

    dyad_ids_0 = data['dyad_ids'] - 1
    predictor = edge_weight[dyad_ids_0]

    if data['num_fixed'] > 0:
        beta_fixed = m.dist.normal(data['prior_fixed_mu'], data['prior_fixed_sigma'],
                                   shape=(data['num_fixed'],), name="beta_fixed")
```

```
        predictor = predictor + m.jnp.dot(data['design_fixed'], beta_fixed)

    # Scale via event lengths/time blocks
    rate = m.jnp.exp(predictor) * data['divisor']

    # Poisson Likelihood
    m.dist.poisson(rate, obs=data['event'], name="event")

m.fit(bi_model_count)
m.summary()
```

R

```
library(BayesianInference)
m <- importBI(platform = "cpu")
jnp <- reticulate::import("jax.numpy")

m$data_on_model <- list(data = data_list_jax)

bi_model_count <- function(data) {
  edge_weight <- m$dist$normal(data$prior_edge_mu, data$prior_edge_sigma,
                               shape=tuple(data$num_edges), name="edge_weight")

  dyad_ids_0 <- data$dyad_ids - 1L
  predictor <- edge_weight[dyad_ids_0]

  if (data$num_fixed > 0) {
    beta_fixed <- m$dist$normal(data$prior_fixed_mu, data$prior_fixed_sigma,
                                shape=tuple(data$num_fixed), name="beta_fixed")
    predictor <- predictor + jnp$dot(data$design_fixed, beta_fixed)
  }

  rate <- jnp$exp(predictor) * data$divisor

  m$dist$poisson(rate, obs=data$event, name="event")
}

m$fit(bi_model_count)
m$summary()
```

Julia

```
using BayesianInference

m = importBI(platform="cpu")
m.data_on_model = Dict("data" => data_list_jax)

@BI function bi_model_count(data)
    edge_weight = m.dist.normal(data["prior_edge_mu"], data["prior_edge_sigma"],
                                shape=(data["num_edges"],), name="edge_weight")

    dyad_ids_0 = data["dyad_ids"] .- 1
    predictor = edge_weight[dyad_ids_0]

    if data["num_fixed"] > 0
        beta_fixed = m.dist.normal(data["prior_fixed_mu"], data["prior_fixed_sigma"],
                                   shape=(data["num_fixed"],), name="beta_fixed")
        predictor = predictor .+ jnp.dot(data["design_fixed"], beta_fixed)
    end

    rate = jnp.exp(predictor) .* data["divisor"]

    m.dist.poisson(rate, obs=data["event"], name="event")
end

m.fit(bi_model_count)
m.summary()
```

Mathematical Details

For count data, we use the sum of Poisson-distributed random variables to define interactions scaling with observation duration.

$$X_{ij}^{(n)} \sim \text{Poisson}(\lambda_{ij}^{(n)} D_{ij}^{(n)})$$

Where the overall rate modifier is described logarithmically:

$$\log(\lambda_{ij}^{(n)}) = \omega_{ij} + ...$$

Where ω_{ij} evaluates to the maximum likelihood estimation counterpart of straightforward event rate counts across cumulative time blocks.

Notes

> **Note**
>
> The Poisson distribution inherently manages the aggregation mechanics allowing the edge weight to serve dynamically scaled estimates independent from disparate observation lengths.

Duration Data Model

General Principles

The duration model simultaneously handles two data metrics: the *duration* of social events and the *frequency* with which they occur. It treats total engaged time mathematically using a composition approach tracking exponential duration periods bounded across a baseline Poisson event initiation rate.

Considerations

> **Note**
>
> - **Priors**: Requires *two* primary priors. The standard edge-weight proportion models the relative event time (ω), while an overarching generalized `rate` component drives the base observation scale.
> - **Complexity**: Fits two separate likelihood distributions (Poisson and Exponential) interacting across shared predictors to map both count frequencies and continuous lengths simultaneously.

Example with simulated data

Python

```python
from BI import bi

m = bi(platform='cpu')
m.data_on_model = {'data': data_list_jax}

def bi_model_duration(data):
```

```
    # 1. Edge weights prior with partial pooling : Set the baseline (Intercept)
    edge_sigma = m.dist.half_normal(2.0, name="edge_sigma")
    edge_weight = m.dist.normal(0.0, edge_sigma, shape=(data['num_edges'],), name="edge_weigh

    # Additional absolute rate parameter strictly positive
    rate_positive = m.dist.half_normal(1.0, shape=(data['num_edges'],), name="rate")

    dyad_ids_0 = data['dyad_ids'] - 1
    ## Build predictor
    predictor = edge_weight[dyad_ids_0]

    # 2. Fixed nodal factor(s)
    beta_fixed = m.dist.normal(0.0, 1.0, shape=(data['num_fixed'],), name="beta_fixed")
    predictor = predictor + m.jnp.dot(data['design_fixed'], beta_fixed)

    # 3. Random nodal effect(s)
    random_group_mu = m.dist.normal(0.0, 1.0, shape=(data['num_random_groups'],), name="rando
    random_group_sigma = m.dist.half_normal(1.0, shape=(data['num_random_groups'],), name="ra

    group_idx_0 = data['random_group_index'] - 1
    beta_random = m.dist.normal(random_group_mu[group_idx_0], random_group_sigma[group_idx_0]

    predictor = predictor + m.jnp.dot(data['design_random'], beta_random)

    # 4. Link function
    p = m.link.inv_logit(predictor)

    lambda_exp = rate_positive[dyad_ids_0] / p
    lambda_pois = rate_positive * data['divisor']

    # 5. Joint likelihood inference
    m.dist.exponential(lambda_exp, obs=data['event'], name="event")
    m.dist.poisson(lambda_pois, obs=data['event_count'], name="event_count")

m.fit(bi_model_duration)
m.summary()
```

R

```
library(reticulate)
library(BayesianInference)
```

```
m <- importBI(platform = "cpu")
jnp <- import("jax.numpy")

m$data_on_model <- list(data = data_list_jax)

bi_model_duration <- function(data) {

  # 1. Edge weights prior with partial pooling : Set the baseline (Intercept)
  edge_sigma <- m$dist$half_normal(2.0, name = "edge_sigma")
  edge_weight <- m$dist$normal(0.0, edge_sigma, shape = tuple(data$num_edges), name = "edge_w

  # Additional absolute rate parameter strictly positive
  rate_positive <- m$dist$half_normal(1.0, shape = tuple(data$num_edges), name = "rate")

  dyad_ids_0 <- data$dyad_ids - 1L
  ## Build predictor
  predictor <- edge_weight[dyad_ids_0]

  # 2. Fixed nodal factor(s)
  beta_fixed <- m$dist$normal(0.0, 1.0, shape = tuple(data$num_fixed), name = "beta_fixed")
  predictor <- predictor + jnp$dot(data$design_fixed, beta_fixed)

  # 3. Random nodal effect(s)
  random_group_mu <- m$dist$normal(0.0, 1.0, shape = tuple(data$num_random_groups), name = "r
  random_group_sigma <- m$dist$half_normal(1.0, shape = tuple(data$num_random_groups), name =

  group_idx_0 <- data$random_group_index - 1L
  beta_random <- m$dist$normal(random_group_mu[group_idx_0], random_group_sigma[group_idx_0]

  predictor <- predictor + jnp$dot(data$design_random, beta_random)

  # 4. Link function
  p <- m$link.inv_logit(predictor)

  lambda_exp <- rate_positive[dyad_ids_0] / p
  lambda_pois <- rate_positive * data$divisor


  # 5. Joint likelihood inference
  m$dist$exponential(lambda_exp, obs = data$event, name = "event")
  m$dist$poisson(lambda_pois, obs = data$event_count, name = "event_count")
```

```
}

m$fit(bi_model_duration)
m$summary()
```

Julia

```
using BayesianInference

m = importBI(platform="cpu")

jnp = m.jax.numpy

m.data_on_model = Dict("data" => data_list_jax)

@BI function bi_model_duration(data)

    # 1. Edge weights prior with partial pooling : Set the baseline (Intercept)
    edge_sigma = m.dist.half_normal(2.0, name="edge_sigma")
    edge_weight = m.dist.normal(0.0, edge_sigma, shape=(data["num_edges"],), name="edge_weigh

    # Additional absolute rate parameter strictly positive
    rate_positive = m.dist.half_normal(1.0, shape=(data["num_edges"],), name="rate")

    dyad_ids_0 = data["dyad_ids"] .- 1
    ## Build predictor
    predictor = edge_weight[dyad_ids_0]

    # 2. Fixed nodal factor(s)
    beta_fixed = m.dist.normal(0.0, 1.0, shape=(data["num_fixed"],), name="beta_fixed")
    predictor = predictor .+ jnp.dot(data["design_fixed"], beta_fixed)

    # 3. Random nodal effect(s)
    random_group_mu = m.dist.normal(0.0, 1.0, shape=(data["num_random_groups"],), name="rando
    random_group_sigma = m.dist.half_normal(1.0, shape=(data["num_random_groups"],), name="ra

    group_idx_0 = data["random_group_index"] .- 1
    beta_random = m.dist.normal(random_group_mu[group_idx_0], random_group_sigma[group_idx_0]

    predictor = predictor .+ jnp.dot(data["design_random"], beta_random)
```

```
    # 4. Link function
    p = m.link.inv_logit(predictor)

    lambda_exp = rate_positive[dyad_ids_0] ./ p
    #lambda_pois = rate_positive .* data["divisor"]


    m.dist.exponential(lambda_exp, obs=data["event"], name="event")
    #m.dist.poisson(lambda_pois, obs=data["event_count"], name="event_count")
end

m.fit(bi_model_duration)
m.summary()
```

Mathematical Details

The mean behavior time μ_{ij} can be recovered from the estimated proportion scale models representing expected duration lengths per sequence generated by overarching absolute rates λ_{ij}.

$$X_{ij}^{(n)} \sim \text{Exponential}(\lambda_{ij}/t_{ij}^{(n)})$$
$$K_{ij} \sim \text{Poisson}(\lambda_{ij} D_{ij})$$

The proportion is generated matching logistic mapping inputs:

$$\text{logit}(t_{ij}^{(n)}) = \omega_{ij} + ...$$

And recovered as expected proportion values.

Notes

> **Note**
>
> Because edge estimates define probability scales while absolute intensity generates sequence frequency, parameter isolation allows complex uncoupled insights such as distinct testing comparing the absolute magnitude of interactions versus simply calculating engaged durations.

Notes

> **Note**
>
> Varying effects for nodes ad dyads could be build using Multi Variate Normal distribution to account for Node-Level and/or dyadic Correlated Effects.

Hart, Jordan, Michael Nash Weiss, Daniel Franks, and Lauren Brent. 2023. "BISoN: A Bayesian Framework for Inference of Social Networks." *Methods in Ecology and Evolution* 14: 2411–20. https://doi.org/https://doi.org/10.1111/2041-210X.14171.