Network-Based Diffusion Analysis

This document provides a unified framework for performing Bayesian analysis of social transmission using Network-Based Diffusion Analysis (NBDA) models from Chimento and Hoppitt (n.d.) original work. Below, we detail the implementation of various models using the BayesForge (**BF**) package via Python, R, and Julia. All examples assume data is loaded using standard `STbayes` structures Chimento and Hoppitt (n.d.).

General Principles

The principle idea behind Network Based Diffusion analysis (NBDA) is that if social transmission is involved in the spread of a novel behavior through a group, then that spread is expected to follow a social network links Hasenjager, Leadbeater, and Hoppitt (2021). The basic model underlying NBDA states that at time t an individual, i, learns the behavior of interest with a specific rate formula.

In principle NBDA can be consider as a survival analysis, so we have the same concepts as in chapter 14:

1. Where the **baseline hazard** (e.g. the hazard when all covariates are zero) is the asocial hazard.
2. Where the **covariate** is the sum of links toward informed individuals (i.e. individuals that acquired the behavior of interest at time $t-1$).
3. Thus the **Hazard Function** which account for the network links weights **covariate** can thus be consider as the social rate of learning the behavior.

Data

The examples below use the NBDA dataset bundled with the **BF** package (adapted from the STbayes package, Chimento and Hoppitt (n.d.)). It contains acquisition events for 50 individuals across 1 trial and their undirected social network.

Python

```python
from BayesForge import bf
import pandas as pd

m = bf(platform='cpu')

events_path  = m.load.NBDA_events(only_path=True)
network_path = m.load.NBDA_network(only_path=True)
events  = pd.read_csv(events_path)   # columns: id, time, t_end, trial
network = pd.read_csv(network_path)  # columns: focal, other, trial, assoc
# Build data_list following the STbayes-compatible structure - see R example below
```

R

```r
library(BayesForge)
library(STbayes)
m <- importBF(platform = "cpu")
jnp <- reticulate::import("jax.numpy")

to_jax <- function(x) {
    if (is.list(x) || typeof(x) == "closure" || is.character(x)) return(x)
    jnp$array(x)
}

events_path  <- m$load$NBDA_events(only_path = TRUE)
network_path <- m$load$NBDA_network(only_path = TRUE)

data_list <- lapply(
    import_user_STb(read.csv(events_path), read.csv(network_path),
                    network_type = "undirected"),
    to_jax
)
m$data_on_model <- list(data = data_list)
```

Julia

```julia
using BayesForge
using DataFrames, CSV
```

```
m = importBF(platform="cpu")

events_path  = m.load.NBDA_events(only_path=true)
network_path = m.load.NBDA_network(only_path=true)
events  = CSV.read(events_path,  DataFrame)
network = CSV.read(network_path, DataFrame)
```

Continuous Time of Acquisition Data Analysis (cTADA)

General Principles

The cTADA model is used when the exact continuous times of behavioural acquisition are known. It estimates a baseline intrinsic learning rate (λ_0) and a social transmission rate (s). The hazard of learning for an individual depends on their innate capacity to learn independently and the additive influence of their informed social connections over time.

Considerations

> **Note**
>
> - **Priors**: We assign Normal priors on the log-scale for λ_0 and s' (e.g., $Normal(-4, 2)$) to ensure strictly positive rates after exponentiation.
> - **Data**: Requires a survival-type data structure detailing the exact `obs_end_time` (when the event occurred or was censored) and `is_event` status.
> - **Networks**: Computes the additive effect of knowledgeable individuals connected to the naive individual across given networks.

Example

Python

```
from BayesForge import bf

m = bf(platform='cpu')
m.data_on_model = {'data': data_list}
```

```
def model_cTADA(data):
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")

    lambda_0 = m.jnp.exp(log_lambda_0_mean)
    s_prime = m.jnp.exp(log_s_prime_mean)

    A_sum = m.jnp.sum(data['A'], axis=0)
    Z_exp = m.jnp.expand_dims(data['Z'], axis=-1)
    A_Z = m.jnp.squeeze(m.jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime * A_Z

    D_exp = m.jnp.expand_dims(data['D'], axis=-1)
    lambda_all = (lambda_0 + soc_term) * D_exp

    # Survival likelihood: accumulate negative hazard before event, log-hazard at event time
    # m.dist.unit(log_factor=total_log_lik, name="cTADA_lik")

m.fit(model_cTADA)
m.summary()
```

R

```
library(BayesForge)
m <- importBF(platform = "cpu")
m$data_on_model <- list(data = data_list)

model_cTADA <- function(data) {
    log_lambda_0_mean <- m$dist$normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean <- m$dist$normal(-4, 2, name="log_s_prime_mean")

    lambda_0 <- jnp$exp(log_lambda_0_mean)
    s_prime <- jnp$exp(log_s_prime_mean)

    # Matrix multiplications to find connected, knowledgeable peers
    A_sum <- jnp$sum(data$A, axis=0L)
    Z_exp <- jnp$expand_dims(data$Z, axis=-1L)
    A_Z <- jnp$squeeze(jnp$matmul(A_sum, Z_exp), axis=-1L)

    soc_term <- s_prime * A_Z
    D_exp <- jnp$expand_dims(data$D, axis=-1L)
```

```
    lambda_all <- (lambda_0 + soc_term) * D_exp

    # ... Vectorized Unit Likelihood executed here ...
}

m$fit(model_cTADA)
m$summary()
```

Julia

```
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)

@BF function model_cTADA(data)
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")

    lambda_0 = jnp.exp(log_lambda_0_mean)
    s_prime = jnp.exp(log_s_prime_mean)

    A_sum = jnp.sum(data["A"], axis=0)
    Z_exp = jnp.expand_dims(data["Z"], axis=-1)
    A_Z = jnp.squeeze(jnp.matmul(A_sum, Z_exp), axis=-1)

    soc_term = s_prime * A_Z
    D_exp = jnp.expand_dims(data["D"], axis=-1)
    lambda_all = (lambda_0 + soc_term) .* D_exp
end

m.fit(model_cTADA)
m.summary()
```

Mathematical Details

Frequentist Formulation

The hazard rate $\lambda_i(t)$ for individual i at time t is typically defined as:

$$\lambda_i(t) = \lambda_0(t) + s \sum_j a_{i,j} z_j(t)$$

Where: - $\lambda_0(t)$ is the baseline asocial learning rate. - s is the transmission rate per unit of connection. - $a_{i,j}$ represents the network tie strength from j to i. - $z_j(t)$ is an indicator of whether j has acquired the behavior at time t.

Bayesian Formulation

$$\log(\lambda_0) \sim \text{Normal}(\mu_0, \sigma_0)$$
$$\log(s') \sim \text{Normal}(\mu_s, \sigma_s)$$

The specific event acquisition follows a Poisson process survival likelihood evaluating negative hazard over uninformed time steps, and a positive log-hazard at the precise acquisition time cut-point.

Notes

> **Note**
>
> The cTADA model processes hazard calculations fully vectorized in JAX by computing matrices of size `(K, T_max, P)` representing individual statuses at all time cut-points, rendering it extremely computationally efficient on modern hardware.

Order of Acquisition Data Analysis (OADA)

General Principles

OADA is used when the precise times of acquisition are unknown, but the sequential *order* in which individuals acquired the behavior is known. Rather than assessing a continuous survival hazard, OADA estimates the probability that the *next* observed learner was individual i out of the pool of all remaining naive candidates.

Considerations

Note

- **Priors**: Only the relative social transmission rate (s') is estimated as an explicitly defined parameter, often configured with a $Normal(-4, 2)$ prior on the log scale.
- **Asocial Variant**: The `OADA_asocial` variant restricts social transmission to 0, leaving the order predicted solely by equal intrinsic chance.
- **Data Masking**: OADA calculates the likelihood function *only* at the sequence steps where a diffusion event is recorded, significantly trimming empty temporal observations.

Example

Python

```python
from BayesForge import bf

m = bf(platform='cpu')
m.data_on_model = {'data': data_list}

def model_OADA(data):
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")
    s_prime = m.jnp.exp(log_s_prime_mean)

    A_sum = m.jnp.sum(data['A'], axis=0)
    Z_exp = m.jnp.expand_dims(data['Z'], axis=-1)
    A_Z = m.jnp.squeeze(m.jnp.matmul(A_sum, Z_exp), axis=-1)

    net_effect_all = s_prime * A_Z
    lambda_all = 1.0 + net_effect_all  # lambda_0 = 1.0 cancels in the ratio

    # P(i learns next) = lambda_i / sum_j(lambda_j) over naive candidates
    # m.dist.unit(log_factor=total_log_lik, name="OADA_lik")

m.fit(model_OADA)
m.summary()
```

R

```r
library(BayesForge)
m <- importBF(platform = "cpu")
m$data_on_model <- list(data = data_list)

model_OADA <- function(data) {
    log_s_prime_mean <- m$dist$normal(-4, 2, name="log_s_prime_mean")
    s_prime <- jnp$exp(log_s_prime_mean)

    A_sum <- jnp$sum(data$A, axis=0L)
    Z_exp <- jnp$expand_dims(data$Z, axis=-1L)
    A_Z <- jnp$squeeze(jnp$matmul(A_sum, Z_exp), axis=-1L)

    net_effect_all <- s_prime * A_Z
    lambda_all <- 1.0 + net_effect_all

    # ... OADA probability fractional evaluation based on naive population ...
}

m$fit(model_OADA)
m$summary()
```

Julia

```julia
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)

@BF function model_OADA(data)
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")
    s_prime = jnp.exp(log_s_prime_mean)

    A_sum = jnp.sum(data["A"], axis=0)
    Z_exp = jnp.expand_dims(data["Z"], axis=-1)
    A_Z = jnp.squeeze(jnp.matmul(A_sum, Z_exp), axis=-1)

    net_effect_all = s_prime * A_Z
    lambda_all = 1.0 .+ net_effect_all
end
```

```
m.fit(model_OADA)
m.summary()
```

Mathematical Details

Frequentist Formulation

The probability that individual i is the next to acquire the behavior is:

$$P(i|Z) = \frac{\lambda_i}{\sum_{j \notin Z} \lambda_j} = \frac{\lambda_0 + s \sum_k a_{i,k} z_k}{\sum_{j \notin Z} (\lambda_0 + s \sum_k a_{j,k} z_k)}$$

Bayesian Formulation

The hazard rate is treated relatively without needing strictly scaled intrinsic times. The fractional structure uses the total active hazard for candidate i divided by the summation of the hazard models for all currently valid elements inside the population sequence.

Notes

> **Note**
>
> Because the time is abstracted out, OADA effectively treats the base intrinsic learning rate (λ_0) as 1.0. The s' parameter is evaluated as a multiple relative to the baseline intrinsic event.

Individual-Level Variables (ILV)

General Principles

The ILV model integrates continuous, boolean, or categorical traits observed on individuals (e.g., age, sex, rank) into the baseline hazard or the social transmission capabilities. This allows testing hypotheses of whether specific traits predispose candidates to learn faster, either asocially or socially.

Considerations

> **Note**
>
> - **Priors**: The β coefficients applied to the ILVs are assigned standard normal priors, e.g., $Normal(0, 1)$, meaning a zero-centered belief about an ILV having any effect.
> - **Categorical Variables**: Transformed by matching the variable's state index and generating individual specific multipliers across combinations of learning trajectories.

Example

Python

```python
from BayesForge import bf

m = bf(platform='cpu')
m.data_on_model = {'data': data_list}

def model_ILV(data):
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  = m.dist.normal(-4, 2, name="log_s_prime_mean")

    # ILV beta coefficients (boolean→asocial, continuous→social, categorical→multiplicative)
    beta_ILVi = m.dist.normal(0, 1, shape=(1,), name="beta_ILVi_bool_ILV")
    beta_ILVs = m.dist.normal(0, 1, name="beta_ILVs_cont_ILV")
    beta_ILVm = m.dist.normal(0, 1, shape=(3,), name="beta_ILVm_cat_ILV")

    lambda_0 = m.jnp.exp(log_lambda_0_mean)
    s_prime  = m.jnp.exp(log_s_prime_mean)

    bool_ILV_i = m.jnp.matmul(data['ILV_bool_ILV'], beta_ILVi)
    cont_ILV_s = data['ILV_cont_ILV'] * beta_ILVs
    cat_ILV_m  = m.jnp.matmul(data['ILV_cat_ILV'], beta_ILVm)

    ind_term = m.jnp.exp(bool_ILV_i)

    A_sum = m.jnp.sum(data['A'], axis=0)
    Z_exp = m.jnp.expand_dims(data['Z'], axis=-1)
    A_Z   = m.jnp.squeeze(m.jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime * A_Z * m.jnp.exp(cont_ILV_s)
```

```
    D_exp      = m.jnp.expand_dims(data['D'], axis=-1)
    lambda_all = m.jnp.exp(cat_ILV_m) * (lambda_0 * ind_term + soc_term) * D_exp
    # m.dist.unit(log_factor=total_log_lik, name="ILV_lik")

m.fit(model_ILV)
m.summary()
```

R

```
library(BayesForge)
m <- importBF(platform = "cpu")
m$data_on_model <- list(data = data_list)

model_ILV <- function(data) {
    log_lambda_0_mean <- m$dist$normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  <- m$dist$normal(-4, 2, name="log_s_prime_mean")

    beta_ILVi <- m$dist$normal(0, 1, shape=tuple(1L), name="beta_ILVi_bool_ILV")
    beta_ILVs <- m$dist$normal(0, 1, name="beta_ILVs_cont_ILV")
    beta_ILVm <- m$dist$normal(0, 1, shape=tuple(3L), name="beta_ILVm_cat_ILV")

    lambda_0 <- jnp$exp(log_lambda_0_mean)
    s_prime  <- jnp$exp(log_s_prime_mean)

    bool_ILV_i <- jnp$matmul(data$ILV_bool_ILV, beta_ILVi)
    cont_ILV_s <- data$ILV_cont_ILV * beta_ILVs
    cat_ILV_m  <- jnp$matmul(data$ILV_cat_ILV, beta_ILVm)

    ind_term <- jnp$exp(bool_ILV_i)

    A_sum <- jnp$sum(data$A, axis=0L)
    Z_exp <- jnp$expand_dims(data$Z, axis=-1L)
    A_Z   <- jnp$squeeze(jnp$matmul(A_sum, Z_exp), axis=-1L)
    soc_term <- s_prime * A_Z * jnp$exp(cont_ILV_s)

    D_exp      <- jnp$expand_dims(data$D, axis=-1L)
    lambda_all <- jnp$exp(cat_ILV_m) * (lambda_0 * ind_term + soc_term) * D_exp
    # m$dist$unit(log_factor=total_log_lik, name="ILV_lik")
}
```

```
m$fit(model_ILV)
m$summary()
```

Julia

```
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)

@BF function model_ILV(data)
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  = m.dist.normal(-4, 2, name="log_s_prime_mean")

    beta_ILVi = m.dist.normal(0, 1, shape=(1,), name="beta_ILVi_bool_ILV")
    beta_ILVs = m.dist.normal(0, 1, name="beta_ILVs_cont_ILV")
    beta_ILVm = m.dist.normal(0, 1, shape=(3,), name="beta_ILVm_cat_ILV")

    lambda_0 = jnp.exp(log_lambda_0_mean)
    s_prime  = jnp.exp(log_s_prime_mean)

    bool_ILV_i = jnp.matmul(data["ILV_bool_ILV"], beta_ILVi)
    cont_ILV_s = data["ILV_cont_ILV"] .* beta_ILVs
    cat_ILV_m  = jnp.matmul(data["ILV_cat_ILV"], beta_ILVm)

    ind_term = jnp.exp(bool_ILV_i)

    A_sum = jnp.sum(data["A"], axis=0)
    Z_exp = jnp.expand_dims(data["Z"], axis=-1)
    A_Z   = jnp.squeeze(jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime .* A_Z .* jnp.exp(cont_ILV_s)

    D_exp      = jnp.expand_dims(data["D"], axis=-1)
    lambda_all = jnp.exp.(cat_ILV_m) .* (lambda_0 .* ind_term .+ soc_term) .* D_exp
end

m.fit(model_ILV)
m.summary()
```

Mathematical Details

Frequentist Formulation

ILVs alter the transmission framework as follows:

$$\lambda_i(t) = \exp(X_i^{\beta}) \times \left[\lambda_0 \exp(X^{\alpha}_{asocial_i}) + s \sum_j a_{i,j} z_j(t) \exp(X^{\gamma}_{social_i}) \right]$$

Where β, α, γ are vectors calculating multiplicative trait influences on global, asocial, and social transmission weights respectively.

Bayesian Formulation

$$\beta_{ILV} \sim \text{Normal}(0, 1)$$

Variables scale multiplicatively.

Notes

> **Note**
>
> The exponential conversion $\exp(V_i \times \beta)$ enforces positive hazard modifiers, ensuring trait influence scales the absolute chance of behavior generation.

Varying Effects (veff)

General Principles

Models containing varying (random) effects assign specific variance structures, typically for individual personalities or repeated behavioral trials across multiple datasets. The `veff` model offsets intrinsic (λ_0) and social (s') rates for each candidate inside the recorded sample dimensions.

Considerations

> **Note**
>
> - **Priors**: Employs hierarchical modeling. Group means $\bar{\lambda}_0$ and $\bar{s}'$ observe normal definitions, while variance σ parameters dictating the individual offsets are given Half-Normal distributions to ensure positively scaled variance matrices.

- **Population Definition**: Relies on accurate array dimensions dynamically obtained from the JAX tracer graph (`Z$shape`) inside the model function.

Example

Python

```
from BayesForge import bf

m = bf(platform='cpu')
m.data_on_model = {'data': data_list}

def model_veff(data):
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")

    # Recover Individual counts from static shape data (for JIT compilation safety)
    P = data['Z'].shape[2]

    sigma_sprime = m.dist.half_normal(1.0, name="sigma_sprime")
    sigma_lambda0 = m.dist.half_normal(1.0, name="sigma_lambda0")

    v_sprime = m.dist.normal(0.0, sigma_sprime, shape=(P,), name="v_sprime")
    v_lambda0 = m.dist.normal(0.0, sigma_lambda0, shape=(P,), name="v_lambda0")

    s_prime  = m.jnp.exp(log_s_prime_mean + v_sprime)   # shape (P,)
    lambda_0 = m.jnp.exp(log_lambda_0_mean + v_lambda0) # shape (P,)

    A_sum = m.jnp.sum(data['A'], axis=0)
    Z_exp = m.jnp.expand_dims(data['Z'], axis=-1)
    A_Z   = m.jnp.squeeze(m.jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime * A_Z  # per-individual s_prime broadcasts over (K, T_max, P)

    D_exp      = m.jnp.expand_dims(data['D'], axis=-1)
    lambda_all = (lambda_0 + soc_term) * D_exp
    # m.dist.unit(log_factor=total_log_lik, name="veff_lik")

m.fit(model_veff)
m.summary()
```

R

```r
library(BayesForge)
m <- importBF(platform = "cpu")
m$data_on_model <- list(data = data_list)

model_veff <- function(data) {
    log_lambda_0_mean <- m$dist$normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean <- m$dist$normal(-4, 2, name="log_s_prime_mean")

    # 1-indexed R extraction. Python matrix shape axis 2 is [[3]]
    P <- data$Z$shape[[3]]

    sigma_sprime <- m$dist$half_normal(1.0, name="sigma_sprime")
    sigma_lambda0 <- m$dist$half_normal(1.0, name="sigma_lambda0")

    v_sprime <- m$dist$normal(0.0, sigma_sprime, shape=tuple(P), name="v_sprime")
    v_lambda0 <- m$dist$normal(0.0, sigma_lambda0, shape=tuple(P), name="v_lambda0")

    s_prime  <- jnp$exp(log_s_prime_mean + v_sprime)   # shape (P,)
    lambda_0 <- jnp$exp(log_lambda_0_mean + v_lambda0) # shape (P,)

    A_sum <- jnp$sum(data$A, axis=0L)
    Z_exp <- jnp$expand_dims(data$Z, axis=-1L)
    A_Z   <- jnp$squeeze(jnp$matmul(A_sum, Z_exp), axis=-1L)
    soc_term <- s_prime * A_Z

    D_exp      <- jnp$expand_dims(data$D, axis=-1L)
    lambda_all <- (lambda_0 + soc_term) * D_exp
    # m$dist$unit(log_factor=total_log_lik, name="veff_lik")
}

m$fit(model_veff)
m$summary()
```

Julia

```julia
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)
```

```
@BF function model_veff(data)
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")

    P = size(data["Z"], 3)

    sigma_sprime = m.dist.half_normal(1.0, name="sigma_sprime")
    sigma_lambda0 = m.dist.half_normal(1.0, name="sigma_lambda0")

    v_sprime = m.dist.normal(0.0, sigma_sprime, shape=(P,), name="v_sprime")
    v_lambda0 = m.dist.normal(0.0, sigma_lambda0, shape=(P,), name="v_lambda0")

    s_prime  = jnp.exp.(log_s_prime_mean .+ v_sprime)
    lambda_0 = jnp.exp.(log_lambda_0_mean .+ v_lambda0)

    A_sum = jnp.sum(data["A"], axis=0)
    Z_exp = jnp.expand_dims(data["Z"], axis=-1)
    A_Z   = jnp.squeeze(jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime .* A_Z

    D_exp      = jnp.expand_dims(data["D"], axis=-1)
    lambda_all = (lambda_0 .+ soc_term) .* D_exp
end

m.fit(model_veff)
m.summary()
```

Mathematical Details

Frequentist Formulation

Usually addressed as Generalized Linear Mixed Models (GLMMs) handling "random effects":

$$\lambda_i = \lambda_0 + \nu_{0,i}$$

$v_{0,i}$ defines the unobserved error parameter assigned to offset individual i.

Bayesian Formulation

$$\sigma_{\lambda 0} \sim \text{HalfNormal}(1.0)$$

$$v_{0,i} \sim \text{Normal}(0.0, \sigma_{\lambda 0})$$

Bayesian representations natively support hierarchical estimations representing specific individuals drawing unique offsets based strictly on the overarching dataset standard deviation profile.

Notes

> **Note**
>
> This model effectively captures personality attributes when measuring overlapping traits across identical individuals, identifying distinct transmission behavior profiles.

Complex Transmission (complex_f)

General Principles

Traditional linear assumption states transmission increases uniformly with more informed connections. The `complex_f` formulation introduces an exponential modifier f governing positive or negative frequency-dependent limits. This allows modeling thresholds where behaviors only transmit after a certain *proportion* of a cohort knows the skill.

Considerations

> **Note**
>
> - **Priors**: Evaluating a network structure requires defining active (knowledgeable) connections and inactive (naive) connections. The parameter `log_f_mean` is treated under a $Normal(0, 1)$ curve.
> - **Alternative parameters**: The `complex_k` shape structure serves as a numerically stable replacement utilizing uniform sigmoid distributions instead of raw exponent variables, but achieves similar non-linear adjustments.

Example

Python

```python
from BayesForge import bf

m = bf(platform='cpu')
m.data_on_model = {'data': data_list}

def model_complex_f(data):
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")
    log_f_mean = m.dist.normal(0, 1, name="log_f_mean")

    lambda_0 = m.jnp.exp(log_lambda_0_mean)
    s_prime = m.jnp.exp(log_s_prime_mean)
    f = m.jnp.exp(log_f_mean)

    # Assess Active/Inactive associations
    active = m.jnp.squeeze(m.jnp.matmul(data['A'], data['Z']), axis=-1)
    inactive = m.jnp.squeeze(m.jnp.matmul(data['A'], (1.0 - data['Zn'])), axis=-1)

    # Scale through frequency exponent
    active_f = m.jnp.power(active, f)
    denom_f = active_f + m.jnp.power(inactive, f)

    # Filter 0/0 exceptions
    frac = m.jnp.where((active + inactive) > 0, active_f / denom_f, 0.0)
    net_effect = m.jnp.sum(frac, axis=0)
    soc_term = s_prime * net_effect

    D_exp      = m.jnp.expand_dims(data['D'], axis=-1)
    lambda_all = (lambda_0 + soc_term) * D_exp
    # m.dist.unit(log_factor=total_log_lik, name="complex_f_lik")

m.fit(model_complex_f)
m.summary()
```

R

```r
library(BayesForge)
m <- importBF(platform = "cpu")
m$data_on_model <- list(data = data_list)
```

```r
model_complex_f <- function(data) {
    log_lambda_0_mean <- m$dist$normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  <- m$dist$normal(-4, 2, name="log_s_prime_mean")
    log_f_mean        <- m$dist$normal(0, 1, name="log_f_mean")

    lambda_0 <- jnp$exp(log_lambda_0_mean)
    s_prime  <- jnp$exp(log_s_prime_mean)
    f        <- jnp$exp(log_f_mean)

    # Expand Z and Zn for matrix multiply against A (N_networks, K, T_max, P, P)
    Z_exp  <- jnp$expand_dims(jnp$expand_dims(data$Z,  axis=-1L), axis=0L)
    Zn_exp <- jnp$expand_dims(jnp$expand_dims(data$Zn, axis=-1L), axis=0L)

    active   <- jnp$squeeze(jnp$matmul(data$A, Z_exp),          axis=-1L)
    inactive <- jnp$squeeze(jnp$matmul(data$A, 1.0 - Zn_exp),   axis=-1L)

    active_f <- jnp$power(active,   f)
    denom_f  <- active_f + jnp$power(inactive, f)
    frac     <- jnp$where((active + inactive) > 0, active_f / denom_f, 0.0)

    net_effect <- jnp$sum(frac, axis=0L)
    soc_term   <- s_prime * net_effect

    D_exp      <- jnp$expand_dims(data$D, axis=-1L)
    lambda_all <- (lambda_0 + soc_term) * D_exp
    # m$dist$unit(log_factor=total_log_lik, name="complex_f_lik")
}

m$fit(model_complex_f)
m$summary()
```

Julia

```julia
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)

@BF function model_complex_f(data)
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean = m.dist.normal(-4, 2, name="log_s_prime_mean")
```

```
    log_f_mean = m.dist.normal(0, 1, name="log_f_mean")
    f = jnp.exp(log_f_mean)

    active = jnp.squeeze(jnp.matmul(data["A"], data["Z"]), axis=-1)
    inactive = jnp.squeeze(jnp.matmul(data["A"], (1.0 .- data["Zn"])), axis=-1)

    active_f = jnp.power(active, f)
    denom_f = active_f .+ jnp.power(inactive, f)

    frac = jnp.where((active .+ inactive) .> 0, active_f ./ denom_f, 0.0)
    net_effect = jnp.sum(frac, axis=0)
end

m.fit(model_complex_f)
m.summary()
```

Mathematical Details

Frequentist Formulation

The hazard function involves fractional frequency components evaluating connected neighbors. Rather than unscaled connection summation:

$$s \times \frac{(\sum_j a_{ij} z_j)^f}{(\sum_j a_{ij} z_j)^f + (\sum_j a_{ij}(1 - z_j))^f}$$

Where f defines the transmission bias exponent. Values of $f > 1$ indicate disproportionate influence from majorities (conformity), and $f < 1$ values support extreme novelty transmission where knowledge readily spreads despite low frequency.

Bayesian Formulation

$$\log(f) \sim \text{Normal}(0, 1)$$

Notes

> **Note**
>
> Because the scale of s' shifts its domain interpretation relative to fractional connections (e.g., s becomes relative to having *all* connections highly informed rather than a single

unit), comparing raw transmission weights directly between standard representations and complex curves yields improper alignment.

Dynamic Networks & Transmission Weights (dynamic_tweights)

General Principles

Some phenomena involve interactions bounded over fluctuating intervals, such as highly motile populations where connections change dynamically. Furthermore, the likelihood that individuals express specific behavioral traits (the "transmission weight") varies over time. The Dynamic networks framework implements functional transformations supporting exact temporal alignments.

Considerations

> **Note**
>
> - **Priors**: Employs the k-shape curve parameterization applied through local normalized bounding: mapping unconstrained standard distributions into $(-1, 1)$ domain mappings via `sigmoid` functionality.
> - **Networks**: Computes interactions tracking T_{max} discrete time occurrences against the spatial coordinates.

Example

Python

```python
from BayesForge import bf

m = bf(platform='cpu')
m.data_on_model = {'data': data_list}

def model_dynamic_tweights(data):
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  = m.dist.normal(-4, 2, name="log_s_prime_mean")
    k_raw   = m.dist.normal(0, 3, name="k_raw")
```

```
    k_shape = 2.0 * m.jax.nn.sigmoid(k_raw) - 1.0

    lambda_0 = m.jnp.exp(log_lambda_0_mean)
    s_prime  = m.jnp.exp(log_s_prime_mean)

    def dini_func(x, k):
        x_t = 2.0 * x - 1.0
        return ((x_t - k * x_t) / (k - 2.0 * k * m.jnp.abs(x_t) + 1.0) + 1.0) / 2.0

    Z_exp  = m.jnp.expand_dims(m.jnp.expand_dims(data['Z'],  axis=-1), axis=0)
    Zn_exp = m.jnp.expand_dims(m.jnp.expand_dims(data['Zn'], axis=-1), axis=0)

    numer = m.jnp.squeeze(m.jnp.matmul(data['A'], Z_exp),        axis=-1)
    denom = numer + m.jnp.squeeze(m.jnp.matmul(data['A'], 1.0 - Zn_exp), axis=-1)

    prop             = m.jnp.where(denom > 0, numer / denom, 0.0)
    dini_transformed = dini_func(prop, k_shape)
    net_effect       = m.jnp.sum(dini_transformed, axis=0)
    soc_term         = s_prime * net_effect

    D_exp      = m.jnp.expand_dims(data['D'], axis=-1)
    lambda_all = (lambda_0 + soc_term) * D_exp
    # m.dist.unit(log_factor=total_log_lik, name="dynamic_networks_tweights_lik")

m.fit(model_dynamic_tweights)
m.summary()
```

R

```
library(BayesForge)
m <- importBF(platform = "cpu")
jax_nn <- reticulate::import("jax.nn")
m$data_on_model <- list(data = data_list)

model_dynamic_tweights <- function(data) {
    log_lambda_0_mean <- m$dist$normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  <- m$dist$normal(-4, 2, name="log_s_prime_mean")
    k_raw   <- m$dist$normal(0, 3, name="k_raw")
    k_shape <- 2.0 * jax_nn$sigmoid(k_raw) - 1.0

    lambda_0 <- jnp$exp(log_lambda_0_mean)
```

```
    s_prime  <- jnp$exp(log_s_prime_mean)

    dini_func <- function(x, k) {
        x_t <- 2.0 * x - 1.0
        ((x_t - k * x_t) / (k - 2.0 * k * jnp$abs(x_t) + 1.0) + 1.0) / 2.0
    }

    Z_exp  <- jnp$expand_dims(jnp$expand_dims(data$Z,  axis=-1L), axis=0L)
    Zn_exp <- jnp$expand_dims(jnp$expand_dims(data$Zn, axis=-1L), axis=0L)

    numer <- jnp$squeeze(jnp$matmul(data$A, Z_exp),          axis=-1L)
    denom <- numer + jnp$squeeze(jnp$matmul(data$A, 1.0 - Zn_exp), axis=-1L)

    prop             <- jnp$where(denom > 0, numer / denom, 0.0)
    dini_transformed <- dini_func(prop, k_shape)
    net_effect       <- jnp$sum(dini_transformed, axis=0L)
    soc_term         <- s_prime * net_effect

    D_exp      <- jnp$expand_dims(data$D, axis=-1L)
    lambda_all <- (lambda_0 + soc_term) * D_exp
    # m$dist$unit(log_factor=total_log_lik, name="dynamic_networks_tweights_lik")
}

m$fit(model_dynamic_tweights)
m$summary()
```

Julia

```
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)

@BF function model_dynamic_tweights(data)
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  = m.dist.normal(-4, 2, name="log_s_prime_mean")
    k_raw    = m.dist.normal(0, 3, name="k_raw")
    k_shape = 2.0 * jax.nn.sigmoid(k_raw) - 1.0

    lambda_0 = jnp.exp(log_lambda_0_mean)
    s_prime  = jnp.exp(log_s_prime_mean)
```

```
    function dini_func(x, k)
        x_t = 2.0 .* x .- 1.0
        return ((x_t .- k .* x_t) ./ (k .- 2.0 .* k .* jnp.abs(x_t) .+ 1.0) .+ 1.0) ./ 2.0
    end

    Z_exp  = jnp.expand_dims(jnp.expand_dims(data["Z"],  axis=-1), axis=0)
    Zn_exp = jnp.expand_dims(jnp.expand_dims(data["Zn"], axis=-1), axis=0)

    numer = jnp.squeeze(jnp.matmul(data["A"], Z_exp),        axis=-1)
    denom = numer .+ jnp.squeeze(jnp.matmul(data["A"], 1.0 .- Zn_exp), axis=-1)

    prop             = jnp.where(denom .> 0, numer ./ denom, 0.0)
    dini_transformed = dini_func(prop, k_shape)
    net_effect       = jnp.sum(dini_transformed, axis=0)
    soc_term         = s_prime .* net_effect

    D_exp      = jnp.expand_dims(data["D"], axis=-1)
    lambda_all = (lambda_0 .+ soc_term) .* D_exp
end

m.fit(model_dynamic_tweights)
m.summary()
```

Mathematical Details

Frequentist Formulation

Dynamic matrices represent adjacency parameters changing per timeline configuration interval t. Thus a_{ij} becomes $a_{ij}(t)$. Similarly, transmission variables map internal traits $w_{ij}(t)$. Combined under the Dini transform fraction, frequency models trace precise granular interactions evaluated uniformly.

Bayesian Formulation

$$k_{raw} \sim \text{Normal}(0, 3)$$

Variables scale within standard sigmoid structures ensuring efficient algorithmic bounds tracking.

Notes

> **Note**
>
> Dynamic transmission parameters require "High-Resolution" data configurations where variables adjust independently across varying observation intervals rather than single global constraints.

Network Uncertainty (posterior_edges)

General Principles

Networks are commonly abstracted from noisy sampling data featuring unequal effort constraints across dyads (missing observations). Rather than employing strict binary thresholds for associations, the `posterior_edges` model implements a multivariate structure predicting internal edge connection likelihoods directly scaled from generated network models (like STRAND or BISoN).

Considerations

> **Note**
>
> - **Priors**: The network connections rely on estimating `edge_logit`, using existing mean array (`logit_edge_mu`) and covariance matrix data (`logit_edge_cov`) extracted from independent network models feeding directly into the diffusion structure.
> - **Networks**: Internally generated A association properties are built by manipulating sampled bounds mapping $P \times P$ matrices through JAX index arrays defining symmetric bidirectional edges.

Example

Python

```
from BayesForge import bf

m = bf(platform='cpu')
```

```
m.data_on_model = {'data': data_list}

def model_posterior_edges(data):
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  = m.dist.normal(-4, 2, name="log_s_prime_mean")

    lambda_0 = m.jnp.exp(log_lambda_0_mean)
    s_prime  = m.jnp.exp(log_s_prime_mean)

    # Sample edge weights from posterior covariance structure
    edge_logit = m.dist.multivariate_normal(
        loc=data['logit_edge_mu'],
        covariance_matrix=data['logit_edge_cov'],
        name="edge_logit"
    )
    w = m.jax.nn.sigmoid(edge_logit)

    # Build symmetric A matrix from sampled edge weights
    N, P = data['N_networks'], data['P']
    A = m.jnp.zeros((N, P, P))
    focal_idx = data['focal_ID'] - 1
    other_idx = data['other_ID'] - 1
    A = A.at[:, focal_idx, other_idx].set(w)
    A = A.at[:, other_idx, focal_idx].set(w)

    A_sum    = m.jnp.sum(A, axis=0)
    Z_exp    = m.jnp.expand_dims(data['Z'], axis=-1)
    A_Z      = m.jnp.squeeze(m.jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime * A_Z

    D_exp      = m.jnp.expand_dims(data['D'], axis=-1)
    lambda_all = (lambda_0 + soc_term) * D_exp
    # m.dist.unit(log_factor=total_log_lik, name="posterior_edges_lik")

m.fit(model_posterior_edges)
m.summary()
```

R

```
library(BayesForge)
m <- importBF(platform = "cpu")
```

```
m$data_on_model <- list(data = data_list)

model_posterior_edges <- function(data) {
    log_lambda_0_mean <- m$dist$normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  <- m$dist$normal(-4, 2, name="log_s_prime_mean")

    lambda_0 <- jnp$exp(log_lambda_0_mean)
    s_prime  <- jnp$exp(log_s_prime_mean)

    edge_logit <- m$dist$multivariate_normal(
        loc = data$logit_edge_mu,
        covariance_matrix = data$logit_edge_cov,
        name = "edge_logit"
    )

    jax <- reticulate::import("jax")
    w <- jax$nn$sigmoid(edge_logit)

    # Build symmetric A from sampled dyadic weights
    N  <- as.integer(data$N_networks)
    P  <- as.integer(data$P)
    A  <- jnp$zeros(tuple(N, P, P))
    focal_idx <- data$focal_ID - 1L
    other_idx <- data$other_ID - 1L
    net_idx   <- jnp$arange(N)[, jnp$newaxis]
    A <- A$at[net_idx, focal_idx, other_idx]$set(w)
    A <- A$at[net_idx, other_idx, focal_idx]$set(w)

    A_sum    <- jnp$sum(A, axis=0L)
    Z_exp    <- jnp$expand_dims(data$Z, axis=-1L)
    A_Z      <- jnp$squeeze(jnp$matmul(A_sum, Z_exp), axis=-1L)
    soc_term <- s_prime * A_Z

    D_exp      <- jnp$expand_dims(data$D, axis=-1L)
    lambda_all <- (lambda_0 + soc_term) * D_exp
    # m$dist$unit(log_factor=total_log_lik, name="posterior_edges_lik")
}

m$fit(model_posterior_edges)
m$summary()
```

Julia

```
using BayesForge

m = importBF(platform="cpu")
m.data_on_model = Dict("data" => data_list)

@BF function model_posterior_edges(data)
    log_lambda_0_mean = m.dist.normal(-4, 2, name="log_lambda_0_mean")
    log_s_prime_mean  = m.dist.normal(-4, 2, name="log_s_prime_mean")

    lambda_0 = jnp.exp(log_lambda_0_mean)
    s_prime  = jnp.exp(log_s_prime_mean)

    edge_logit = m.dist.multivariate_normal(
        loc=data["logit_edge_mu"], covariance_matrix=data["logit_edge_cov"], name="edge_logi
    )
    w = jax.nn.sigmoid(edge_logit)

    N, P = data["N_networks"], data["P"]
    A = jnp.zeros((N, P, P))
    focal_idx = data["focal_ID"] .- 1
    other_idx = data["other_ID"] .- 1
    A = A.at[:, focal_idx, other_idx].set(w)
    A = A.at[:, other_idx, focal_idx].set(w)

    A_sum    = jnp.sum(A, axis=0)
    Z_exp    = jnp.expand_dims(data["Z"], axis=-1)
    A_Z      = jnp.squeeze(jnp.matmul(A_sum, Z_exp), axis=-1)
    soc_term = s_prime .* A_Z

    D_exp      = jnp.expand_dims(data["D"], axis=-1)
    lambda_all = (lambda_0 .+ soc_term) .* D_exp
end

m.fit(model_posterior_edges)
m.summary()
```

Mathematical Details

Frequentist Formulation

Standard applications represent associations identically tracking uniform interactions across specific point coordinates (a_{ij}). Values lacking sample observations resolve missing attributes as strictly void constraints reducing baseline interaction.

Bayesian Formulation

$$a_{ij} \sim \text{MultivariateNormal}(\mu_{dyad}, \Sigma_{dyad})$$

Integrating network boundaries resolves the joint uncertainty structure allowing accurate estimates mapping population distributions rather than raw sample limits, dramatically eliminating false negative assertions representing untethered population elements.

Notes

> **Note**
>
> This model prevents the severe over-estimation of intrinsic (λ_0) properties observed frequently executing standardized unweighted networks tracking under-sampled elements.

Chimento, Michael, and William Hoppitt. n.d. "STbayes: An r Package for Creating, Fitting and Understanding Bayesian Models of Social Transmission." *Methods in Ecology and Evolution* n/a (n/a). https://doi.org/https://doi.org/10.1111/2041-210x.70228.

Hasenjager, Matthew J., Ellouise Leadbeater, and William Hoppitt. 2021. "Detecting and Quantifying Social Transmission Using Network-Based Diffusion Analysis." *Journal of Animal Ecology* 90 (1): 8–26. https://doi.org/https://doi.org/10.1111/1365-2656.13307.