dibs

Lars Lorch

Feb 27, 2024

CONTENTS:

1 Inference **3**

1.1 dibs . . . 3

1.1.1 dibs.inference package . . . 3

1.1.2 dibs.models package . . . 12

1.1.3 dibs.kernel module . . . 18

1.1.4 dibs.metrics module . . . 19

1.1.5 dibs.target module . . . 20

1.1.6 dibs.graph_utils module . . . 23

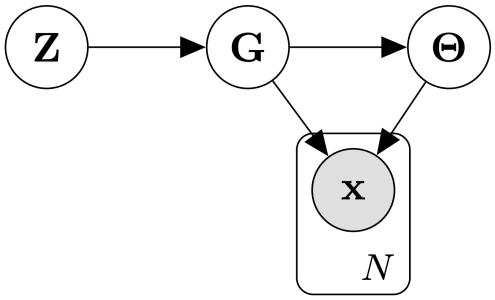
2 Indices and tables **25**

Python Module Index **27**

Index **29**

This is the Python JAX implementation for **DiBS: Differentiable Bayesian Structure Learning** (Lorch et al., 2021). This documentation specifies the API and interaction of the components of the inference pipeline. The entire code is written in JAX to leverage just-in-time compilation, automatic differentation, vectorized operations, and hardware acceleration.

DiBS translates learning $p(G|D)$ and $p(G, \Theta|D)$ over causal Bayesian networks (G, Θ) into inference over the continuous latent posterior densities $p(Z|D)$ and $p(Z, \Theta|D)$, respectively. This extended generative assumptions is illustrated by the following graphical model:



Since we can efficiently estimate the scores $\nabla_Z \log p(Z|D)$ and $\nabla_Z \log p(Z, \Theta|D)$, general-purpose approximate inference methods apply off-the-shelf. Further information and experimental results can be found in the paper.

CHAPTER

ONE

INFERENCE

In this repository, DiBS inference is implemented with the particle variational inference method *Stein Variational Gradient Descent* (SVGD) (Liu and Wang, 2016). To generate samples from $p(Z|D)$, we randomly initialize a set of particles $\{Z_m\}$ and specify some kernel k and step size η. Then, we repeatedly apply the following update **in parallel** for $m = 1$ to M **until convergence**:

$$Z_m \leftarrow Z_m + \eta\phi(Z_m) \text{ where } \phi(\cdot) := \frac{1}{M}\sum_{k=1}^{M} k(Z_k, \cdot)\nabla_{Z_k} \log p(Z_k|D) + \nabla_{Z_k} k(Z_k, \cdot)$$

where at each step $\nabla_{Z_k} \log p(Z_k|D)$ is estimated for each Z_k using the REINFORCE trick or Gumbel-softmax reparameterization. The analogous procedure applies when sampling from $p(Z, \Theta|D)$, where SVGD jointly transports particles Z and Θ.

1.1 dibs

1.1.1 dibs.inference package

DiBS

class dibs.inference.**DiBS**(**, x, interv_mask, log_graph_prior, log_joint_prob, alpha_linear=0.05, beta_linear=1.0, tau=1.0, n_grad_mc_samples=128, n_acyclicity_mc_samples=32, grad_estimator_z='reparam', score_function_baseline=0.0, latent_prior_std=None, verbose=False*)

This class implements the backbone for DiBS, i.e. all gradient estimators and sampling components. Any inference method in the DiBS framework should inherit from this class.

Parameters

- **x** (*ndarray*) – matrix of shape `[n_observations, n_vars]` of i.i.d. observations of the variables
- **interv_mask** (*ndarray*) – binary matrix of shape `[n_observations, n_vars]` indicating whether a given variable was intervened upon in a given sample (intervention = 1, no intervention = 0)
- **log_graph_prior** (*callable*) – function implementing prior $\log p(G)$ of soft adjacency matrix of edge probabilities. For example: `unnormalized_log_prob_soft()` or usually bound in e.g. `log_graph_prior()`
- **log_joint_prob** (*callable*) – function implementing joint likelihood $\log p(\Theta, D|G)$ of parameters and observations given the discrete graph adjacency matrix For example: *dibs.models.LinearGaussian.interventional_log_joint_prob()*. When inferring the

marginal posterior $p(G|D)$ via a closed-form marginal likelihood $\log p(D|G)$, the same function signature has to be satisfied (simply ignoring Θ)

- **alpha_linear** (*float*) – slope of of linear schedule for inverse temperature α of sigmoid in latent graph model $p(G|Z)$
- **beta_linear** (*float*) – slope of of linear schedule for inverse temperature β of constraint penalty in latent prio $p(Z)$
- **tau** (*float*) – constant Gumbel-softmax temperature parameter
- **n_grad_mc_samples** (*int*) – number of Monte Carlo samples in gradient estimator for likelihood term $p(\Theta, D|G)$
- **n_acyclicity_mc_samples** (*int*) – number of Monte Carlo samples in gradient estimator for acyclicity constraint
- **grad_estimator_z** (*str*) – gradient estimator ∇_Z of expectation over $p(G|Z)$; choices: `score` or `reparam`
- **score_function_baseline** (*float*) – scale of additive baseline in score function (REINFORCE) estimator; `score_function_baseline == 0.0` corresponds to not using a baseline
- **latent_prior_std** (*float*) – standard deviation of Gaussian prior over Z; defaults to `1/sqrt(k)`

constraint_gumbel(*single_z*, *single_eps*, *t*)

Evaluates continuous acyclicity constraint using Gumbel-softmax instead of Bernoulli samples

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **single_eps** (*ndarray*) – i.i.d. Logistic noise of shape `[d, d]` for Gumbel-softmax
- **t** (*int*) – step

Returns constraint value of shape `[1,]`

edge_log_probs(*z*, *t*)

Edge log probabilities encoded by latent representation

Parameters

- **z** (*ndarray*) – latent tensors Z `[..., d, k, 2]`
- **t** (*int*) – step

Returns tuple of tensors `[..., d, d]`, `[..., d, d]` corresponding to `log(p)` and `log(1-p)`

edge_probs(*z*, *t*)

Edge probabilities encoded by latent representation

Parameters

- **z** (*ndarray*) – latent tensors Z `[..., d, k, 2]`
- **t** (*int*) – step

Returns edge probabilities of shape `[..., d, d]`

eltwise_grad_latent_log_prob(*gs*, *single_z*, *t*)

Gradient of log likelihood of generative graph model w.r.t. Z i.e. $\nabla_Z \log p(G|Z)$ Batched over samples of G given a single Z.

Parameters

- **gs** (*ndarray*) – batch of graph matrices `[n_graphs, d, d]`
- **single_z** (*ndarray*) – latent variable `[d, k, 2]`
- **t** (*int*) – step

Returns batch of gradients of shape `[n_graphs, d, k, 2]`

eltwise_grad_latent_prior(*zs*, *subkeys*, *t*)

Computes batch of estimators for the score $\nabla_Z \log p(Z)$ with

$$\log p(Z) = -\beta(t) E_{p(G|Z)}[h(G)] + \log \mathcal{N}(Z) + \log f(Z)$$

where h is the acyclicity constraint and *f(Z)* is additional DAG prior factor computed inside `dibs.inference.DiBS.log_graph_prior_particle`.

Parameters

- **zs** (*ndarray*) – single latent tensor `[n_particles, d, k, 2]`
- **subkeys** (*ndarray*) – batch of rng keys `[n_particles, ...]`

Returns batch of gradients of shape `[n_particles, d, k, 2]`

eltwise_grad_theta_likelihood(*zs*, *thetas*, *t*, *subkeys*)

Computes batch of estimators for the score $\nabla_\Theta \log p(\Theta, D|Z)$, i.e. w.r.t the conditional distribution parameters. Uses the same $G \sim p(G|Z)$ samples for expectations in numerator and denominator.

This does not use $\nabla_G \log p(\Theta, D|G)$ and is hence applicable when the gradient w.r.t. the adjacency matrix is not defined (as e.g. for the BGe score). Analogous to `eltwise_grad_z_likelihood` but gradient w.r.t Θ instead of Z

Parameters

- **zs** (*ndarray*) – batch of latent tensors Z of shape `[n_particles, d, k, 2]`
- **thetas** (*Any*) – batch of parameter PyTree with `n_mc_samples` as leading dim

Returns batch of gradients in form of `thetas` PyTree with `n_particles` as leading dim

eltwise_grad_z_likelihood(*zs*, *thetas*, *baselines*, *t*, *subkeys*)

Computes batch of estimators for score $\nabla_Z \log p(\Theta, D|Z)$ Selects corresponding estimator used for the term $\nabla_Z E_{p(G|Z)}[p(\Theta, D|G)]$ and executes it in batch.

Parameters

- **zs** (*ndarray*) – batch of latent tensors Z `[n_particles, d, k, 2]`
- **thetas** (*Any*) – batch of parameters PyTree with `n_particles` as leading dim
- **baselines** (*ndarray*) – array of score function baseline values of shape `[n_particles, ]`

Returns tuple batch of (gradient estimates, baselines) of shapes `[n_particles, d, k, 2]`, `[n_particles, ]`

eltwise_log_joint_prob(*gs*, *single_theta*, *rng*)

Joint likelihood $\log p(\Theta, D|G)$ batched over samples of G

Parameters

- **gs** (*ndarray*) – batch of graphs `[n_graphs, d, d]`
- **single_theta** (*Any*) – single parameter PyTree
- **rng** (*ndarray*) – for mini-batching `x` potentially

Returns batch of logprobs of shape `[n_graphs, ]`

grad_constraint_gumbel(*single_z*, *key*, *t*)

Reparameterization estimator for the gradient $\nabla_Z E_{p(G|Z)}[h(G)]$ where h is the acyclicity constraint penalty function.

Since h is differentiable w.r.t. G, always uses the Gumbel-softmax / concrete distribution reparameterization trick.

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **key** (*ndarray*) – rng
- **t** (*int*) – step

Returns gradient of shape `[d, k, 2]`

grad_theta_likelihood(*single_z*, *single_theta*, *t*, *subk*)

Computes Monte Carlo estimator for the score $\nabla_\Theta \log p(\Theta, D|Z)$

Uses hard samples of G, but a soft reparameterization like for ∇_Z is also possible. Uses the same $G \sim p(G|Z)$ samples for expectations in numerator and denominator.

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **single_theta** (*Any*) – single parameter PyTree
- **t** (*int*) – step
- **subk** (*ndarray*) – rng key

Returns parameter gradient PyTree

grad_z_likelihood_gumbel(*single_z*, *single_theta*, *single_sf_baseline*, *t*, *subk*)

Reparameterization estimator for the score $\nabla_Z \log p(\Theta, D|Z)$ sing the Gumbel-softmax / concrete distribution reparameterization trick. Uses the same $G \sim p(G|Z)$ samples for expectations in numerator and denominator.

This **does** require a well-defined gradient $\nabla_G \log p(\Theta, D|G)$ and is hence not applicable when the gradient w.r.t. the adjacency matrix is not defined for Gumbel-relaxations of the discrete adjacency matrix. Any (marginal) likelihood expressible as a function of `g[:, j]` and `theta` , e.g. using the vector of (possibly soft) parent indicators as a mask, satisfies this.

Examples are: `dibs.models.LinearGaussian` and `dibs.models.DenseNonlinearGaussian` See also e.g. http://proceedings.mlr.press/v108/zheng20a/zheng20a.pdf

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **single_theta** (*Any*) – single parameter PyTree
- **single_sf_baseline** (*ndarray*) – `[1, ]`
- **t** (*int*) – step
- **subk** (*ndarray*) – rng key

Returns tuple of gradient, baseline `[d, k, 2], [1, ]`

grad_z_likelihood_score_function(*single_z*, *single_theta*, *single_sf_baseline*, *t*, *subk*)

Score function estimator (aka REINFORCE) for the score $\nabla_Z \log p(\Theta, D|Z)$ Uses the same $G \sim p(G|Z)$ samples for expectations in numerator and denominator.

This does not use $\nabla_G \log p(\Theta, D|G)$ and is hence applicable when the gradient w.r.t. the adjacency matrix is not defined (as e.g. for the BGe score).

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **single_theta** (*Any*) – single parameter PyTree
- **single_sf_baseline** (*ndarray*) – `[1, ]`
- **t** (*int*) – step
- **subk** (*ndarray*) – rng key

Returns tuple of gradient, baseline `[d, k, 2], [1, ]`

latent_log_prob(*single_g*, *single_z*, *t*)
Log likelihood of generative graph model

Parameters

- **single_g** (*ndarray*) – single graph adjacency matrix `[d, d]`
- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **t** (*int*) – step

Returns log likelihood $logp(G|Z)$ of shape `[1,]`

log_graph_prior_particle(*single_z*, *t*)
Computes $\log p(G)$ component of $\log p(Z)$, i.e. not the contraint or Gaussian prior term, but the DAG belief.

The log prior $\log p(G)$ is evaluated with edge probabilities $G_\alpha(Z)$ given Z.

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **t** (*int*) – step

Returns log prior graph probability`log p(G_{alpha}(Z))` of shape `[1,]`

log_joint_prob_soft(*single_z*, *single_theta*, *eps*, *t*, *subk*)
This is the composition of $\log p(\Theta, D|G)$ (Gumbel-softmax graph sample given Z)

Parameters

- **single_z** (*ndarray*) – single latent tensor `[d, k, 2]`
- **single_theta** (*Any*) – single parameter PyTree
- **eps** (*ndarray*) – i.i.d Logistic noise of shape `[d, d]`
- **t** (*int*) – step
- **subk** (*ndarray*) – rng key

Returns logprob of shape `[1, ]`

particle_to_g_lim(z)
Returns G corresponding to $\alpha = \infty$ for particles z

Parameters **z** (*ndarray*) – latent variables `[..., d, k, 2]`

Returns graph adjacency matrices of shape `[..., d, d]`

particle_to_hard_graph(*z*, *eps*, *t*)
Bernoulli sample of G using probabilities implied by latent `z`

Parameters

- **z** (*ndarray*) – a single latent tensor Z of shape `[d, k, 2]`
- **eps** (*ndarray*) – random i.i.d. Logistic(0,1) noise of shape `[d, d]`
- **t** (*int*) – step

Returns Gumbel-max (hard) sample of adjacency matrix `[d, d]`

particle_to_soft_graph(*z*, *eps*, *t*)
Gumbel-softmax / concrete distribution using Logistic(0,1) samples `eps`

Parameters

- **z** (*ndarray*) – a single latent tensor Z of shape `[d, k, 2]`\`
- **eps** (*ndarray*) – random i.i.d. Logistic(0,1) noise of shape `[d, d]`
- **t** (*int*) – step

Returns Gumbel-softmax sample of adjacency matrix [d, d]

sample_g(*p*, *subk*, *n_samples*)
Sample Bernoulli matrix according to matrix of probabilities

Parameters

- **p** (*ndarray*) – matrix of probabilities `[d, d]`
- **n_samples** (*int*) – number of samples
- **subk** (*ndarray*) – rng key

Returns an array of matrices sampled according to `p` of shape `[n_samples, d, d]`

visualize_callback(*ipython=True*, *save_path=None*)
Returns callback function for visualization of particles during inference updates

Parameters

- **ipython** (*bool*) – set to `True` when running in a jupyter notebook
- **save_path** (*str*) – path to save plotted images to

Returns callback

SVGD

class dibs.inference.**MarginalDiBS**(**, x, graph_model, likelihood_model, interv_mask=None, kernel=<class 'dibs.kernel.AdditiveFrobeniusSEKernel'>, kernel_param=None, optimizer='rmsprop', optimizer_param=None, alpha_linear=1.0, beta_linear=1.0, tau=1.0, n_grad_mc_samples=128, n_acyclicity_mc_samples=32, grad_estimator_z='score', score_function_baseline=0.0, latent_prior_std=None, verbose=False*)
Bases: *dibs.inference.dibs.DiBS*

This class implements Stein Variational Gradient Descent (SVGD) (Liu and Wang, 2016) for DiBS inference (Lorch et al., 2021) of the marginal DAG posterior $p(G|D)$. For joint inference of $p(G, \Theta|D)$, use the analogous class *JointDiBS*.

An SVGD update of tensor v is defined as

$\phi(v) \propto \sum_u k(v,u) \nabla_u \log p(u) + \nabla_u k(u,v)$

Parameters

- **x** (*ndarray*) – observations of shape `[n_observations, n_vars]`
- **interv_mask** (*ndarray, optional*) – binary matrix of shape `[n_observations, n_vars]` indicating whether a given variable was intervened upon in a given sample (intervention = 1, no intervention = 0)
- **graph_model** – Model defining the prior $\log p(G)$ underlying the inferred posterior. Object *has to implement one method*: `unnormalized_log_prob_soft` Example: *ErdosReniDAGDistribution*
- **likelihood_model** – Model defining the marginal likelihood $\log p(D|G)$ underlying the inferred posterior. Object *has to implement one method*: `interventional_log_marginal_prob` Example: *BGe*
- **kernel** – Class of kernel. *Has to implement the method* `eval(u, v)`. Example: *AdditiveFrobeniusSEKernel*
- **kernel_param** (*dict*) – kwargs to instantiate `kernel`
- **optimizer** (*str*) – optimizer identifier
- **optimizer_param** (*dict*) – kwargs to instantiate `optimizer`
- **alpha_linear** (*float*) – slope of of linear schedule for inverse temperature α of sigmoid in latent graph model $p(G|Z)$
- **beta_linear** (*float*) – slope of of linear schedule for inverse temperature β of constraint penalty in latent prior $p(Z)$
- **tau** (*float*) – constant Gumbel-softmax temperature parameter
- **n_grad_mc_samples** (*int*) – number of Monte Carlo samples in gradient estimator for likelihood term $p(\Theta, D|G)$
- **n_acyclicity_mc_samples** (*int*) – number of Monte Carlo samples in gradient estimator for acyclicity constraint
- **grad_estimator_z** (*str*) – gradient estimator ∇_Z of expectation over $p(G|Z)$; choices: `score` or `reparam`
- **score_function_baseline** (*float*) – scale of additive baseline in score function (REINFORCE) estimator; `score_function_baseline == 0.0` corresponds to not using a baseline
- **latent_prior_std** (*float*) – standard deviation of Gaussian prior over Z; defaults to `1/sqrt(k)`

get_empirical(g)

Converts batch of binary (adjacency) matrices into *empirical* particle distribution where mixture weights correspond to counts/occurrences

Parameters **g** (*ndarray*) – batch of graph samples `[n_particles, d, d]` with binary values

Returns particle distribution of graph samples and associated log probabilities

Return type *ParticleDistribution*

get_mixture(g)

Converts batch of binary (adjacency) matrices into *mixture* particle distribution, where mixture weights correspond to unnormalized target (i.e. posterior) probabilities

Parameters **g** (*ndarray*) – batch of graph samples `[n_particles, d, d]` with binary values

Returns particle distribution of graph samples and associated log probabilities

Return type *ParticleDistribution*

sample(*, *key*, *n_particles*, *steps*, *n_dim_particles=None*, *callback=None*, *callback_every=None*)
Use SVGD with DiBS to sample `n_particles` particles G from the marginal posterior $p(G|D)$ as defined by the BN model `self.inference_model`

Parameters

- **key** (*ndarray*) – prng key
- **n_particles** (*int*) – number of particles to sample
- **steps** (*int*) – number of SVGD steps performed
- **n_dim_particles** (*int*) – latent dimensionality k of particles $Z = \{U, V\}$ with $U, V \in \mathbb{R}^{k \times d}$. Default is `n_vars`
- **callback** – function to be called every `callback_every` steps of SVGD.
- **callback_every** – if `None`, `callback` is only called after particle updates have finished

Returns batch of samples $G \sim p(G|D)$ of shape `[n_particles, n_vars, n_vars]`

class dibs.inference.**JointDiBS**(*, *x*, *graph_model*, *likelihood_model*, *interv_mask=None*, *kernel=<class 'dibs.kernel.JointAdditiveFrobeniusSEKernel'>*, *kernel_param=None*, *optimizer='rmsprop'*, *optimizer_param=None*, *alpha_linear=0.05*, *beta_linear=1.0*, *tau=1.0*, *n_grad_mc_samples=128*, *n_acyclicity_mc_samples=32*, *grad_estimator_z='reparam'*, *score_function_baseline=0.0*, *latent_prior_std=None*, *verbose=False*)

Bases: *dibs.inference.dibs.DiBS*

This class implements Stein Variational Gradient Descent (SVGD) (Liu and Wang, 2016) for DiBS inference (Lorch et al., 2021) of the marginal DAG posterior $p(G|D)$. For marginal inference of $p(G|D)$, use the analogous class *MarginalDiBS*.

An SVGD update of tensor v is defined as

$\phi(v) \propto \sum_u k(v, u) \nabla_u \log p(u) + \nabla_u k(u, v)$

Parameters

- **x** (*ndarray*) – observations of shape `[n_observations, n_vars]`
- **interv_mask** (*ndarray, optional*) – binary matrix of shape `[n_observations, n_vars]` indicating whether a given variable was intervened upon in a given sample (intervention = 1, no intervention = 0)
- **graph_model** – Model defining the prior $\log p(G)$ underlying the inferred posterior. Object *has to implement one method*: `unnormalized_log_prob_soft` Example: *ErdosReniDAGDistribution*
- **likelihood_model** – Model defining the joint likelihood $\log p(\Theta, D|G) = \log p(\Theta|G) + \log p(D|G, \Theta)$ underlying the inferred posterior. Object *has to implement one method*: `interventional_log_joint_prob` Example: *LinearGaussian*
- **kernel** – Class of kernel. *Has to implement the method* `eval(u, v)`. Example: *JointAdditiveFrobeniusSEKernel*
- **kernel_param** (*dict*) – kwargs to instantiate `kernel`
- **optimizer** (*str*) – optimizer identifier

- **optimizer_param** (*dict*) – kwargs to instantiate `optimizer`
- **alpha_linear** (*float*) – slope of of linear schedule for inverse temperature α of sigmoid in latent graph model $p(G|Z)$
- **beta_linear** (*float*) – slope of of linear schedule for inverse temperature β of constraint penalty in latent prior $p(Z)$
- **tau** (*float*) – constant Gumbel-softmax temperature parameter
- **n_grad_mc_samples** (*int*) – number of Monte Carlo samples in gradient estimator for likelihood term $p(\Theta, D|G)$
- **n_acyclicity_mc_samples** (*int*) – number of Monte Carlo samples in gradient estimator for acyclicity constraint
- **grad_estimator_z** (*str*) – gradient estimator ∇_Z of expectation over $p(G|Z)$; choices: `score` or `reparam`
- **score_function_baseline** (*float*) – scale of additive baseline in score function (REINFORCE) estimator; `score_function_baseline == 0.0` corresponds to not using a baseline
- **latent_prior_std** (*float*) – standard deviation of Gaussian prior over Z; defaults to `1/sqrt(k)`

get_empirical(*g*, *theta*)

Converts batch of binary (adjacency) matrices and parameters into *empirical* particle distribution where mixture weights correspond to counts/occurrences

Parameters

- **g** (*ndarray*) – batch of graph samples `[n_particles, d, d]` with binary values
- **theta** (*Any*) – PyTree with leading dim `n_particles`

Returns particle distribution of graph and parameter samples and associated log probabilities

Return type *ParticleDistribution*

get_mixture(*g*, *theta*)

Converts batch of binary (adjacency) matrices and particles into *mixture* particle distribution, where mixture weights correspond to unnormalized target (i.e. posterior) probabilities

Parameters

- **g** (*ndarray*) – batch of graph samples `[n_particles, d, d]` with binary values
- **theta** (*Any*) – PyTree with leading dim `n_particles`

Returns particle distribution of graph and parameter samples and associated log probabilities

Return type *ParticleDistribution*

sample(*, *key*, *n_particles*, *steps*, *n_dim_particles=None*, *callback=None*, *callback_every=None*)

Use SVGD with DiBS to sample `n_particles` particles (G, Θ) from the joint posterior $p(G, \Theta|D)$ as defined by the BN model `self.likelihood_model`

Parameters

- **key** (*ndarray*) – prng key
- **n_particles** (*int*) – number of particles to sample
- **steps** (*int*) – number of SVGD steps performed

- **n_dim_particles** (*int*) – latent dimensionality k of particles $Z = \{U, V\}$ with $U, V \in \mathbb{R}^{k \times d}$. Default is `n_vars`
- **callback** – function to be called every `callback_every` steps of SVGD.
- **callback_every** – if `None`, `callback` is only called after particle updates have finished

Returns batch of samples $G, \Theta \sim p(G, \Theta | D)$

Return type tuple of shape (`[n_particles, n_vars, n_vars]`, `PyTree`) where `PyTree` has leading dimension `n_particles`

1.1.2 dibs.models package

Graph models

class dibs.models.**ErdosReniDAGDistribution**(*n_vars*, *n_edges_per_node=2*)

Randomly oriented Erdos-Reni random graph model with i.i.d. edge probability. The pmf is defined as

$p(G) \propto p^e (1-p)^{\binom{d}{2} - e}$

where e denotes the total number of edges in G and p is chosen to satisfy the requirement of sampling `n_edges_per_node` edges per node in expectation.

Parameters

- **n_vars** (*int*) – number of variables in DAG
- **n_edges_per_node** (*int*) – number of edges sampled per variable in expectation

sample_G(*key*, *return_mat=False*)

Samples DAG

Parameters

- **key** (*ndarray*) – rng
- **return_mat** (*bool*) – if `True`, returns adjacency matrix of shape `[n_vars, n_vars]`

Returns DAG

Return type `iGraph.graph` / `jnp.array`

unnormalized_log_prob(*, *g*)

Computes $\log p(G)$ up the normalization constant

Parameters **g** (*iGraph.graph*) – graph

Returns unnormalized log probability of G

unnormalized_log_prob_single(*, *g*, *j*)

Computes $\log p(G_j)$ up the normalization constant

Parameters

- **g** (*iGraph.graph*) – graph
- **j** (*int*) – node index:

Returns unnormalized log probability of node family of j

unnormalized_log_prob_soft(*, *soft_g*)

Computes $\log p(G)$ up the normalization constant where G is the matrix of edge probabilities

Parameters **soft_g** (*ndarray*) – graph adjacency matrix, where entries may be probabilities and not necessarily 0 or 1

Returns unnormalized log probability corresponding to edge probabilities in G

class dibs.models.**ScaleFreeDAGDistribution**(*n_vars*, *verbose=False*, *n_edges_per_node=2*)

Randomly-oriented scale-free random graph with power-law degree distribution. The pmf is defined as

$p(G) \propto \prod_j (1 + \deg(j))^{-3}$

where $\deg(j)$ denotes the in-degree of node j

Parameters

- **n_vars** (*int*) – number of variables in DAG
- **n_edges_per_node** (*int*) – number of edges sampled per variable

sample_G(*key*, *return_mat=False*)

Samples DAG

Parameters

- **key** (*ndarray*) – rng
- **return_mat** (*bool*) – if `True`, returns adjacency matrix of shape `[n_vars, n_vars]`

Returns DAG

Return type `iGraph.graph` / `jnp.array`

unnormalized_log_prob(*, *g*)

Computes $\log p(G)$ up the normalization constant

Parameters **g** (*iGraph.graph*) – graph

Returns unnormalized log probability of G

unnormalized_log_prob_single(*, *g*, *j*)

Computes $\log p(G_j)$ up the normalization constant

Parameters

- **g** (*iGraph.graph*) – graph
- **j** (*int*) – node index:

Returns unnormalized log probability of node family of j

unnormalized_log_prob_soft(*, *soft_g*)

Computes $\log p(G)$ up the normalization constant where G is the matrix of edge probabilities

Parameters **soft_g** (*ndarray*) – graph adjacency matrix, where entries may be probabilities and not necessarily 0 or 1

Returns unnormalized log probability corresponding to edge probabilities in G

Observational models

class dibs.models.**BGe**(*, *n_vars*, *mean_obs=None*, *alpha_mu=None*, *alpha_lambd=None*)

Linear Gaussian BN model corresponding to linear structural equation model (SEM) with additive Gaussian noise. Uses Normal-Wishart conjugate parameter prior to allow for closed-form marginal likelihood $\log p(D|G)$ and thus allows inference of the marginal posterior $p(G|D)$

For details on the closed-form expression, refer to

- Geiger et al. (2002): https://projecteuclid.org/download/pdf_1/euclid.aos/1035844981
- Kuipers et al. (2014): https://projecteuclid.org/download/suppdf_1/euclid.aos/1407420013

The default arguments imply commonly-used default hyperparameters for mean and precision of the Normal-Wishart and assume a diagonal parameter matrix T. Inspiration for the implementation was drawn from https://bitbucket.org/jamescussens/pygobnilp/src/master/pygobnilp/scoring.py

This implementation uses properties of the determinant to make the computation of the marginal likelihood `jax.jit`-compilable and `jax.grad`-differentiable by remaining well-defined for soft relaxations of the graph.

Parameters

- **n_vars** (*int*) – number of variables (nodes in the graph)
- **mean_obs** (*ndarray, optional*) – mean parameter of Normal
- **alpha_mu** (*float, optional*) – precision parameter of Normal
- **alpha_lambd** (*float, optional*) – degrees of freedom parameter of Wishart

interventional_log_marginal_prob(*g*, *_*, *x*, *interv_targets*, *rng*)

Computes interventional marginal likelihood $\log p(D|G)$ in closed-form; `jax.jit`-compatible

To unify the function signatures for the marginal and joint inference classes *MarginalDiBS* and *JointDiBS*, this marginal likelihood is defined with dummy `theta` inputs as `_`, i.e., like a joint likelihood

Parameters

- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`. Entries must be binary and of type `jnp.int32`
- **_** –
- **x** (*ndarray*) – observational data of shape `[n_observations, n_vars]`
- **interv_targets** (*ndarray*) – indicator mask of interventions of shape `[n_observations, n_vars]`
- **rng** (*ndarray*) – rng; skeleton for minibatching (TBD)

Returns BGe score of shape `[1,]`

log_marginal_likelihood(*, *g*, *x*, *interv_targets*)

Computes BGe marginal likelihood $\log p(D|G)$ in closed-form; `jax.jit`-compatible

Parameters

- **g** (*ndarray*) – adjacency matrix of shape `[d, d]`
- **x** (*ndarray*) – observations of shape `[N, d]`
- **interv_targets** (*ndarray*) – boolean mask of shape `[N, d]` of whether or not a node was intervened upon in a given sample. Intervened nodes are ignored in likelihood computation

Returns BGe Score

class dibs.models.**LinearGaussian**(*, *n_vars*, *obs_noise=0.1*, *mean_edge=0.0*, *sig_edge=1.0*, *min_edge=0.5*)

Linear Gaussian BN model corresponding to linear structural equation model (SEM) with additive Gaussian noise.

Each variable distributed as Gaussian with mean being the linear combination of its parents weighted by a Gaussian parameter vector (i.e., with Gaussian-valued edges). The noise variance at each node is equal by default, which implies the causal structure is identifiable.

Parameters

- **n_vars** (*int*) – number of variables (nodes in the graph)
- **obs_noise** (*float, optional*) – variance of additive observation noise at nodes
- **mean_edge** (*float, optional*) – mean of Gaussian edge weight
- **sig_edge** (*float, optional*) – std dev of Gaussian edge weight
- **min_edge** (*float, optional*) – minimum linear effect of parent on child

get_theta_shape(*, *n_vars*)

Returns tree shape of the parameters of the linear model

Parameters **n_vars** (*int*) – number of variables in model

Returns PyTree of parameter shape

interventional_log_joint_prob(*g*, *theta*, *x*, *interv_targets*, *rng*)

Computes interventional joint likelihood $\log p(\Theta, D|G)$

Parameters

- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`
- **theta** (*ndarray*) – parameter matrix of shape `[n_vars, n_vars]`
- **x** (*ndarray*) – observational data of shape `[n_observations, n_vars]`
- **interv_targets** (*ndarray*) – indicator mask of interventions of shape `[n_observations, n_vars]`
- **rng** (*ndarray*) – rng; skeleton for minibatching (TBD)

Returns log prob of shape `[1,]`

log_likelihood(*, *x*, *theta*, *g*, *interv_targets*)

Computes likelihood $p(D|G, \Theta)$. In this model, the noise per observation and node is additive and Gaussian.

Parameters

- **x** (*ndarray*) – observations of shape `[n_observations, n_vars]`
- **theta** (*ndarray*) – parameters of shape `[n_vars, n_vars]`
- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`
- **interv_targets** (*ndarray*) – binary intervention indicator vector of shape `[n_observations, n_vars]`

Returns log prob

log_prob_parameters(*, *theta*, *g*)

Computes parameter prior $\log p(\Theta|G)$ In this model, the parameter prior is Gaussian.

Parameters

- **theta** (*ndarray*) – parameter matrix of shape `[n_vars, n_vars]`
- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`

Returns log prob

sample_obs(*, *key*, *n_samples*, *g*, *theta*, *toporder=None*, *interv=None*)
Samples `n_samples` observations given graph `g` and parameters `theta`

Parameters

- **key** (*ndarray*) – rng
- **n_samples** (*int*) – number of samples
- **g** (*igraph.Graph*) – graph
- **theta** (*Any*) – parameters
- **interv** (*dict*) – intervention specification of the form `{intervened node : clamp value}`

Returns observation matrix of shape `[n_samples, n_vars]`

sample_parameters(*, *key*, *n_vars*, *n_particles=0*, *batch_size=0*)
Samples batch of random parameters given dimensions of graph from $p(\Theta|G)$

Parameters

- **key** (*ndarray*) – rng
- **n_vars** (*int*) – number of variables in BN
- **n_particles** (*int*) – number of parameter particles sampled
- **batch_size** (*int*) – number of batches of particles being sampled

Returns Parameters `theta` of shape `[batch_size, n_particles, n_vars, n_vars]`, dropping dimensions equal to 0

class dibs.models.**DenseNonlinearGaussian**(*, *n_vars*, *hidden_layers*, *obs_noise=0.1*, *sig_param=1.0*, *activation='relu'*, *bias=True*)

Nonlinear Gaussian BN model corresponding to a nonlinaer structural equation model (SEM) with additive Gaussian noise.

Each variable distributed as Gaussian with mean parameterized by a dense neural network (MLP) whose weights and biases are sampled from a Gaussian prior. The noise variance at each node is equal by default.

Refer to http://proceedings.mlr.press/v108/zheng20a/zheng20a.pdf

Parameters

- **n_vars** (*int*) – number of variables (nodes in the graph)
- **hidden_layers** (*tuple*) – list of integers specifying the number of layers as well as their widths. For example: `[8, 8]` would correspond to 2 hidden layers with 8 neurons
- **obs_noise** (*float, optional*) – variance of additive observation noise at nodes
- **sig_param** (*float, optional*) – std dev of Gaussian parameter prior
- **activation** (*str, optional*) – identifier for activation function. Choices: `sigmoid`, `tanh`, `relu`, `leakyrelu`

get_theta_shape(*, *n_vars*)
Returns tree shape of the parameters of the neural networks

Parameters **n_vars** (*int*) – number of variables in model

Returns PyTree of parameter shape

`interventional_log_joint_prob`(*g*, *theta*, *x*, *interv_targets*, *rng*)
Computes interventional joint likelihood $\log p(\Theta, D|G)$

Parameters

- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`
- **theta** (*Any*) – parameter PyTree
- **x** (*ndarray*) – observational data of shape `[n_observations, n_vars]`
- **interv_targets** (*ndarray*) – indicator mask of interventions of shape `[n_observations, n_vars]`
- **rng** (*ndarray*) – rng; skeleton for minibatching (TBD)

Returns log prob of shape `[1,]`

`log_likelihood`(*, *x*, *theta*, *g*, *interv_targets*)
Computes likelihood $p(D|G, \Theta)$. In this model, the noise per observation and node is additive and Gaussian.

Parameters

- **x** (*ndarray*) – observations of shape `[n_observations, n_vars]`
- **theta** (*Any*) – parameters PyTree
- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`
- **interv_targets** (*ndarray*) – binary intervention indicator vector of shape `[n_vars, ]`

Returns log prob

`log_prob_parameters`(*, *theta*, *g*)
Computes parameter prior $\log p(\Theta|G)$ In this model, the prior over weights and biases is zero-centered Gaussian.

Parameters

- **theta** (*Any*) – parameter pytree
- **g** (*ndarray*) – graph adjacency matrix of shape `[n_vars, n_vars]`

Returns log prob

`sample_obs`(*, *key*, *n_samples*, *g*, *theta*, *toporder=None*, *interv=None*)
Samples `n_samples` observations given graph `g` and parameters `theta` by doing single forward passes in topological order

Parameters

- **key** (*ndarray*) – rng
- **n_samples** (*int*) – number of samples
- **g** (*igraph.Graph*) – graph
- **theta** (*Any*) – parameters
- **interv** (*dict*) – intervention specification of the form `{intervened node : clamp value}`

Returns observation matrix of shape `[n_samples, n_vars]`

sample_parameters(*, *key*, *n_vars*, *n_particles=0*, *batch_size=0*)
Samples batch of random parameters given dimensions of graph from $p(\Theta|G)$

Parameters

- **key** (*ndarray*) – rng
- **n_vars** (*int*) – number of variables in BN
- **n_particles** (*int*) – number of parameter particles sampled
- **batch_size** (*int*) – number of batches of particles being sampled

Returns Parameter PyTree with leading dimension(s) `batch_size` and/or `n_particles`, dropping either dimension when equal to 0

1.1.3 dibs.kernel module

class dibs.kernel.**AdditiveFrobeniusSEKernel**(*, *h=20.0*, *scale=1.0*)
Squared exponential kernel defined as

$k(Z, Z') = \text{scale} \cdot \exp(-\frac{1}{h}||Z - Z'||_F^2)$

Parameters

- **h** (*float*) – bandwidth parameter
- **scale** (*float*) – scale parameter

eval(*, *x*, *y*)
Evaluates kernel function

Parameters

- **x** (*ndarray*) – any shape `[...]`
- **y** (*ndarray*) – any shape `[...]`, but same as `x`

Returns kernel value of shape `[1,]`

class dibs.kernel.**JointAdditiveFrobeniusSEKernel**(*, *h_latent=5.0*, *h_theta=500.0*, *scale_latent=1.0*, *scale_theta=1.0*)
Squared exponential kernel defined as

$k((Z, \Theta), (Z', \Theta')) = \text{scale}_z \cdot \exp(-\frac{1}{h_z}||Z - Z'||_F^2) + \text{scale}_\theta \cdot \exp(-\frac{1}{h_\theta}||\Theta - \Theta'||_F^2)$

Parameters

- **h_latent** (*float*) – bandwidth parameter for Z term
- **h_theta** (*float*) – bandwidth parameter for Θ term
- **scale_latent** (*float*) – scale parameter for Z term
- **scale_theta** (*float*) – scale parameter for Θ term

eval(*, *x_latent*, *x_theta*, *y_latent*, *y_theta*)
Evaluates kernel function k(x, y)

Parameters

- **x_latent** (*ndarray*) – any shape `[...]`
- **x_theta** (*Any*) – any PyTree of `jnp.array` tensors
- **y_latent** (*ndarray*) – any shape `[...]`, but same as `x_latent`

- **y_theta** (*Any*) – any PyTree of `jnp.array` tensors, but same as `x_theta`

Returns kernel value of shape `[1,]`

1.1.4 dibs.metrics module

class dibs.metrics.**ParticleDistribution**(*logp: Any*, *g: Any*, *theta: Optional[Any] = None*)

NamedTuple for structuring sampled particles (G, Θ) (or G) and their assigned log probabilities

Parameters

- **logp** (*ndarray*) – vector of log probabilities or weights of shape `[M, ]`
- **g** (*ndarray*) – batch of graph adjacency matrix of shape `[M, d, d]`
- **theta** (*ndarray*) – batch of parameter PyTrees with leading dimension `M`

g: Any
Alias for field number 1

logp: Any
Alias for field number 0

theta: Any
Alias for field number 2

dibs.metrics.**expected_edges**(***, *dist*)

Computes expected number of edges, defined as

expected edges$(p) := \sum_G p(G|D)|\text{edges}(G)|$

Parameters **dist** (*dibs.metrics.ParticleDistribution*) – particle distribution

Returns expected number of edges `[1, ]`

dibs.metrics.**expected_shd**(***, *dist*, *g*)

Computes expected structural hamming distance metric, defined as

expected SHD$(p, G^*) := \sum_G p(G|D)\text{SHD}(G, G^*)$

Parameters

- **dist** (*dibs.metrics.ParticleDistribution*) – particle distribution
- **g** (*ndarray*) – ground truth adjacency matrix of shape `[d, d]`

Returns expected SHD `[1, ]`

dibs.metrics.**neg_ave_log_likelihood**(***, *dist*, *eltwise_log_likelihood*, *x*)

Computes neg. ave log likelihood for a joint posterior over (G, Θ), defined as

neg. LL$(p, G^*) := -\sum_G \int_\Theta p(G, \Theta|D)p(D^{\text{test}}|G, \Theta)$

Parameters

- **dist** (*dibs.metrics.ParticleDistribution*) – particle distribution
- **eltwise_log_likelihood** (*callable*) – function evaluting the log likelihood $p(D|G, \Theta)$ for a batch of graph samples given a data set of held-out observations; must satisfy the signature `[:, d, d], PyTree(leading dim :), [N, d] -> [:,]`
- **x** (*ndarray*) – held-out observations of shape `[N, d]`

Returns neg. ave log likelihood metric of shape `[1,]`

dibs.metrics.**neg_ave_log_marginal_likelihood**(*, *dist*, *eltwise_log_marginal_likelihood*, *x*)
Computes neg. ave log marginal likelihood for a marginal posterior over G, defined as

neg. MLL$(p, G^*) := -\sum_G p(G|D)p(D^{\text{test}}|G)$

Parameters

- **dist** (*dibs.metrics.ParticleDistribution*) – particle distribution
- **eltwise_log_marginal_likelihood** (*callable*) – function evaluting the marginal log likelihood $p(D|G)$ for a batch of graph samples given a data set of held-out observations; must satisfy the signature `[:, d, d], [N, d] -> [:,]`
- **x** (*ndarray*) – held-out observations of shape `[N, d]`

Returns neg. ave log marginal likelihood metric of shape `[1,]`

dibs.metrics.**pairwise_structural_hamming_distance**(*, *x*, *y*)
Computes pairwise Structural Hamming distance, i.e. the number of edge insertions, deletions or flips in order to transform one graph to another This means, edge reversals do not double count, and that getting an undirected edge wrong only counts 1

Parameters

- **x** (*ndarray*) – batch of adjacency matrices [N, d, d]
- **y** (*ndarray*) – batch of adjacency matrices [M, d, d]

Returns matrix of shape `[N, M]` where elt `i,j` is SHD(`x[i]`, `y[j]`)

dibs.metrics.**threshold_metrics**(*, *dist*, *g*)
Computes various threshold metrics (e.g. ROC, precision-recall, ...)

Parameters

- **dist** (*dibs.metrics.ParticleDistribution*) – sampled particle distribution
- **g** (*ndarray*) – ground truth adjacency matrix of shape `[d, d]`

Returns dict of metrics

1.1.5 dibs.target module

class dibs.target.**Data**(*passed_key: Any*, *n_vars: int*, *n_observations: int*, *n_ho_observations: int*, *g: Any*, *theta: Any*, *x: Any*, *x_ho: Any*, *x_interv: Any*)
NamedTuple for structuring simulated synthetic data and their ground truth generative model

Parameters

- **passed_key** (*ndarray*) – `jax.random` key passed *into* the function generating this object
- **n_vars** (*int*) – number of variables in model
- **n_observations** (*int*) – number of observations in `x` and used to perform inference
- **n_ho_observations** (*int*) – number of held-out observations in `x_ho` and elements of `x_interv` used for evaluation
- **g** (*ndarray*) – ground truth DAG
- **theta** (*Any*) – ground truth parameters
- **x** (*ndarray*) – i.i.d observations from the model of shape `[n_observations, n_vars]`
- **x_ho** (*ndarray*) – i.i.d observations from the model of shape `[n_ho_observations, n_vars]`

- **x_interv** (*list*) – list of (interv dict, i.i.d observations)

g: Any
Alias for field number 4

n_ho_observations: int
Alias for field number 3

n_observations: int
Alias for field number 2

n_vars: int
Alias for field number 1

passed_key: Any
Alias for field number 0

theta: Any
Alias for field number 5

x: Any
Alias for field number 6

x_ho: Any
Alias for field number 7

x_interv: Any
Alias for field number 8

dibs.target.**make_graph_model**(*, *n_vars*, *graph_prior_str*, *edges_per_node=2*)
Instantiates graph model

Parameters

- **n_vars** (*int*) – number of variables in graph
- **graph_prior_str** (*str*) – specifier for random graph model; choices: `er`, `sf`
- **edges_per_node** (*int*) – number of edges per node (in expectation when applicable)

Returns Object representing graph model. For example *ErdosReniDAGDistribution* or *ScaleFreeDAGDistribution*

dibs.target.**make_linear_gaussian_equivalent_model**(*, *key*, *n_vars=20*, *graph_prior_str='sf'*, *bge_mean_obs=None*, *bge_alpha_mu=None*, *bge_alpha_lambd=None*, *obs_noise=0.1*, *mean_edge=0.0*, *sig_edge=1.0*, *min_edge=0.5*, *n_observations=100*, *n_ho_observations=100*)
Samples a synthetic linear Gaussian BN instance with Bayesian Gaussian equivalent (BGe) marginal likelihood as inference model to weight each DAG in an MEC equally

By marginalizing out the parameters, the BGe model does not allow inferring the parameters Θ.

Parameters

- **key** (*ndarray*) – rng key
- **n_vars** (*int*) – number of variables i
- **n_observations** (*int*) – number of iid observations of variables
- **n_ho_observations** (*int*) – number of iid held-out observations of variables
- **graph_prior_str** (*str*) – graph prior (`er` or `sf`)
- **bge_mean_obs** (*float*) – BGe score prior mean parameter of Normal

- **bge_alpha_mu** (*float*) – BGe score prior precision parameter of Normal
- **bge_alpha_lambd** (*float*) – BGe score prior effective sample size (degrees of freedom parameter of Wishart)
- **obs_noise** (*float*) – observation noise
- **mean_edge** (*float*) – edge weight mean
- **sig_edge** (*float*) – edge weight stddev
- **min_edge** (*float*) – min edge weight enforced by constant shift of sampled parameter

Returns BGe inference model and observations from a linear Gaussian generative process

Return type tuple(*BGe*, *Data*)

dibs.target.**make_linear_gaussian_model**(**, key, n_vars=20, graph_prior_str='sf', obs_noise=0.1, mean_edge=0.0, sig_edge=1.0, min_edge=0.5, n_observations=100, n_ho_observations=100*)

Samples a synthetic linear Gaussian BN instance

Parameters

- **key** (*ndarray*) – rng key
- **n_vars** (*int*) – number of variables
- **n_observations** (*int*) – number of iid observations of variables
- **n_ho_observations** (*int*) – number of iid held-out observations of variables
- **graph_prior_str** (*str*) – graph prior (*er* or *sf*)
- **obs_noise** (*float*) – observation noise
- **mean_edge** (*float*) – edge weight mean
- **sig_edge** (*float*) – edge weight stddev
- **min_edge** (*float*) – min edge weight enforced by constant shift of sampled parameter

Returns linear Gaussian inference model and observations from a linear Gaussian generative process

Return type tuple(*LinearGaussian*, *Data*)

dibs.target.**make_nonlinear_gaussian_model**(**, key, n_vars=20, graph_prior_str='sf', obs_noise=0.1, sig_param=1.0, hidden_layers=(5,), n_observations=100, n_ho_observations=100*)

Samples a synthetic nonlinear Gaussian BN instance where the local conditional distributions are parameterized by fully-connected neural networks.

Parameters

- **key** (*ndarray*) – rng key
- **n_vars** (*int*) – number of variables
- **n_observations** (*int*) – number of iid observations of variables
- **n_ho_observations** (*int*) – number of iid held-out observations of variables
- **graph_prior_str** (*str*) – graph prior (*er* or *sf*)
- **obs_noise** (*float*) – observation noise
- **sig_param** (*float*) – stddev of the BN parameters, i.e. here the neural net weights and biases

- **hidden_layers** (*tuple*) – list of ints specifying the hidden layer (sizes) of the neural nets parameterizatin the local condtitionals

Returns nonlinear Gaussian inference model and observations from a nonlinear Gaussian generative process

Return type tuple(*DenseNonlinearGaussian*, Target)

dibs.target.**make_synthetic_bayes_net**(*, *key*, *n_vars*, *graph_model*, *generative_model*, *n_observations=100*, *n_ho_observations=100*, *n_intervention_sets=10*, *perc_intervened=0.1*)

Returns an instance of Target for evaluation of a method on a ground truth synthetic causal Bayesian network

Parameters

- **key** (*ndarray*) – rng key
- **n_vars** (*int*) – number of variables
- **graph_model** (*Any*) – graph model object. For example: *ErdosReniDAGDistribution*
- **generative_model** (*Any*) – BN model object for generating the observations. For example: *LinearGaussian*
- **n_observations** (*int*) – number of observations generated for posterior inference
- **n_ho_observations** (*int*) – number of held-out observations generated for evaluation
- **n_intervention_sets** (*int*) – number of different interventions considered overall for generating interventional data
- **perc_intervened** (*float*) – percentage of nodes intervened upon (clipped to 0) in an intervention.

Returns synthetic ground truth generative DAG and parameters as well observations sampled from the model

Return type *Data*

1.1.6 dibs.graph_utils module

dibs.graph_utils.**acyclic_constr_nograd**(*mat*, *n_vars*)

Differentiable acyclicity constraint from Yu et al. (2019) http://proceedings.mlr.press/v97/yu19a/yu19a.pdf

Parameters

- **mat** (*ndarray*) – graph adjacency matrix of shape [n_vars, n_vars]
- **n_vars** (*int*) – number of variables, to allow for jax.jit-compilation

Returns constraint value [1,]

dibs.graph_utils.**adjmat_to_str**(*mat*, *max_len=40*)

Converts binary adjacency matrix to human-readable string

Parameters

- **mat** (*ndarray*) – graph adjacency matrix
- **max_len** (*int*) – maximum length of string

Returns human readable description of edges in adjacency matrix

Return type str

dibs.graph_utils.**elwise_acyclic_constr_nograd**(*mat*, *n_vars*)
Vectorized version of acyclic_constr_nograd. Takes similar arguments as acyclic_constr_nograd but with additional array axes over which acyclic_constr_nograd is mapped.

Original documentation:

Differentiable acyclicity constraint from Yu et al. (2019) http://proceedings.mlr.press/v97/yu19a/yu19a.pdf

Args: mat (ndarray): graph adjacency matrix of shape `[n_vars, n_vars]` n_vars (int): number of variables, to allow for `jax.jit`-compilation

Returns: constraint value `[1, ]`

dibs.graph_utils.**graph_to_mat**(*g*)
Returns adjacency matrix of `ig.Graph` object

Parameters **g** (*igraph.Graph*) – graph

Returns adjacency matrix

Return type ndarray

dibs.graph_utils.**mat_is_dag**(*mat*)
Returns `True` iff adjacency matrix represents a DAG

Parameters **mat** (*ndarray*) – graph adjacency matrix

Returns `True` iff `mat` represents a DAG

Return type bool

dibs.graph_utils.**mat_to_graph**(*mat*)
Returns `ig.Graph` object for adjacency matrix

Parameters **mat** (*ndarray*) – adjacency matrix

Returns graph

Return type igraph.Graph

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

dibs.graph_utils, 23
dibs.inference, 3
dibs.kernel, 18
dibs.metrics, 19
dibs.models, 12
dibs.target, 20

INDEX

A

acyclic_constr_nograd() (*in module dibs.graph_utils*), 23
AdditiveFrobeniusSEKernel (*class in dibs.kernel*), 18
adjmat_to_str() (*in module dibs.graph_utils*), 23

B

BGe (*class in dibs.models*), 14

C

constraint_gumbel() (*dibs.inference.DiBS method*), 4

D

Data (*class in dibs.target*), 20
DenseNonlinearGaussian (*class in dibs.models*), 16
DiBS (*class in dibs.inference*), 3
dibs.graph_utils
    module, 23
dibs.inference
    module, 3
dibs.kernel
    module, 18
dibs.metrics
    module, 19
dibs.models
    module, 12
dibs.target
    module, 20

E

edge_log_probs() (*dibs.inference.DiBS method*), 4
edge_probs() (*dibs.inference.DiBS method*), 4
eltwise_grad_latent_log_prob() (*dibs.inference.DiBS method*), 4
eltwise_grad_latent_prior() (*dibs.inference.DiBS method*), 5
eltwise_grad_theta_likelihood() (*dibs.inference.DiBS method*), 5
eltwise_grad_z_likelihood() (*dibs.inference.DiBS method*), 5
eltwise_log_joint_prob() (*dibs.inference.DiBS method*), 5
elwise_acyclic_constr_nograd() (*in module dibs.graph_utils*), 23
ErdosReniDAGDistribution (*class in dibs.models*), 12
eval() (*dibs.kernel.AdditiveFrobeniusSEKernel method*), 18
eval() (*dibs.kernel.JointAdditiveFrobeniusSEKernel method*), 18
expected_edges() (*in module dibs.metrics*), 19
expected_shd() (*in module dibs.metrics*), 19

G

g (*dibs.metrics.ParticleDistribution attribute*), 19
g (*dibs.target.Data attribute*), 21
get_empirical() (*dibs.inference.JointDiBS method*), 11
get_empirical() (*dibs.inference.MarginalDiBS method*), 9
get_mixture() (*dibs.inference.JointDiBS method*), 11
get_mixture() (*dibs.inference.MarginalDiBS method*), 9
get_theta_shape() (*dibs.models.DenseNonlinearGaussian method*), 16
get_theta_shape() (*dibs.models.LinearGaussian method*), 15
grad_constraint_gumbel() (*dibs.inference.DiBS method*), 6
grad_theta_likelihood() (*dibs.inference.DiBS method*), 6
grad_z_likelihood_gumbel() (*dibs.inference.DiBS method*), 6
grad_z_likelihood_score_function() (*dibs.inference.DiBS method*), 6
graph_to_mat() (*in module dibs.graph_utils*), 24

I

interventional_log_joint_prob() (*dibs.models.DenseNonlinearGaussian method*), 17
interventional_log_joint_prob() (*dibs.models.LinearGaussian method*), 15
interventional_log_marginal_prob() (*dibs.models.BGe method*), 14

J

JointAdditiveFrobeniusSEKernel (*class in dibs.kernel*), 18
JointDiBS (*class in dibs.inference*), 10

L

latent_log_prob() (*dibs.inference.DiBS method*), 7
LinearGaussian (*class in dibs.models*), 14
log_graph_prior_particle() (*dibs.inference.DiBS method*), 7
log_joint_prob_soft() (*dibs.inference.DiBS method*), 7
log_likelihood() (*dibs.models.DenseNonlinearGaussian method*), 17
log_likelihood() (*dibs.models.LinearGaussian method*), 15
log_marginal_likelihood() (*dibs.models.BGe method*), 14
log_prob_parameters() (*dibs.models.DenseNonlinearGaussian method*), 17
log_prob_parameters() (*dibs.models.LinearGaussian method*), 15
logp (*dibs.metrics.ParticleDistribution attribute*), 19

M

make_graph_model() (*in module dibs.target*), 21
make_linear_gaussian_equivalent_model() (*in module dibs.target*), 21
make_linear_gaussian_model() (*in module dibs.target*), 22
make_nonlinear_gaussian_model() (*in module dibs.target*), 22
make_synthetic_bayes_net() (*in module dibs.target*), 23
MarginalDiBS (*class in dibs.inference*), 8
mat_is_dag() (*in module dibs.graph_utils*), 24
mat_to_graph() (*in module dibs.graph_utils*), 24
module
    dibs.graph_utils, 23
    dibs.inference, 3
    dibs.kernel, 18
    dibs.metrics, 19
    dibs.models, 12
    dibs.target, 20

N

n_ho_observations (*dibs.target.Data attribute*), 21
n_observations (*dibs.target.Data attribute*), 21
n_vars (*dibs.target.Data attribute*), 21
neg_ave_log_likelihood() (*in module dibs.metrics*), 19
neg_ave_log_marginal_likelihood() (*in module dibs.metrics*), 19

P

pairwise_structural_hamming_distance() (*in module dibs.metrics*), 20
particle_to_g_lim() (*dibs.inference.DiBS method*), 7
particle_to_hard_graph() (*dibs.inference.DiBS method*), 7
particle_to_soft_graph() (*dibs.inference.DiBS method*), 8
ParticleDistribution (*class in dibs.metrics*), 19
passed_key (*dibs.target.Data attribute*), 21

S

sample() (*dibs.inference.JointDiBS method*), 11
sample() (*dibs.inference.MarginalDiBS method*), 10
sample_g() (*dibs.inference.DiBS method*), 8
sample_G() (*dibs.models.ErdosReniDAGDistribution method*), 12
sample_G() (*dibs.models.ScaleFreeDAGDistribution method*), 13
sample_obs() (*dibs.models.DenseNonlinearGaussian method*), 17
sample_obs() (*dibs.models.LinearGaussian method*), 16
sample_parameters() (*dibs.models.DenseNonlinearGaussian method*), 17
sample_parameters() (*dibs.models.LinearGaussian method*), 16
ScaleFreeDAGDistribution (*class in dibs.models*), 13

T

theta (*dibs.metrics.ParticleDistribution attribute*), 19
theta (*dibs.target.Data attribute*), 21
threshold_metrics() (*in module dibs.metrics*), 20

U

unnormalized_log_prob() (*dibs.models.ErdosReniDAGDistribution method*), 12
unnormalized_log_prob() (*dibs.models.ScaleFreeDAGDistribution method*), 13
unnormalized_log_prob_single() (*dibs.models.ErdosReniDAGDistribution method*), 12
unnormalized_log_prob_single() (*dibs.models.ScaleFreeDAGDistribution method*), 13
unnormalized_log_prob_soft() (*dibs.models.ErdosReniDAGDistribution method*), 12
unnormalized_log_prob_soft() (*dibs.models.ScaleFreeDAGDistribution method*), 13

V

visualize_callback() (*dibs.inference.DiBS method*), 8

X

x (*dibs.target.Data attribute*), 21
x_ho (*dibs.target.Data attribute*), 21
x_interv (*dibs.target.Data attribute*), 21