Stochastic Variational Inference using Pyro

Stochastic VI recap

Maximize ELBO:

$$\psi^* = \max_{\psi} \mathbb{E}_{q_\psi(z)}[\log p(x, z)] - H(q_\psi)$$

Stochastic VI recap

Draw a sample z to approximate the gradient of the expectation:

$$\nabla_\gamma \mathbb{E}_{q_\gamma(z)}[\log p(x, z) - \log q_\gamma(z)].$$

- REINFORCE (score function estimator).
- Reparameterization.

What is PPL?

- High-level programming language designed for probabilistic modeling by providing explicit mechanism to represent stochasticity (random variables).
- Most languages provide their own modeling language (e.g., JAGS, Stan) or piggy-back on top of existing language like Python (e.g., PyMC, Pyro).
- Abstracts away the difficulties of designing and developing inference methods from modeling.

Previously, changing the model meant re-writing inference code.

What is PPL?

Examples of PPLs:

- BUGS/JAGS: Gibbs sampling based inference engine.
- Stan/PyMC: Hamiltonian Monte Carlo sampling based inference engine.
- Tensorflow probability: Tensorflow backend.
- Pyro: PyTorch backend with primary support for VI.
- NumPyro: NumPy/JAX backend for Pyro with better support for HMC (faster).
- Blang: factor graphs + particle filter/MCMC inference engine.
- Turing.jl: Julia; supports MCMC and VI.

More comprehensive list:
https://en.wikipedia.org/wiki/Probabilistic_programming.

What is PPL?

Languages that utilize HMC or VI typically are built on top of an autograd engine. For example, Pyro on top of PyTorch and Stan has their own backend written in C++.

More customized PPLs are better suited for specific problems. For example, Blang for discrete and combinatorially structured random variables. BUGS/JAGS where conditionals are available.

Pyro

- A Probabilistic programming language (PPL) originally developed by Uber AI with PyTorch backend. Currently maintained by Broad Institute.
- Primarily supports stochastic VI but also offers HMC (sampling based) inference engine.
- The modeling language builds on top of Python syntax.

Pyro

Given a probabilistic model with latent variables denoted z, observations x, and model parameters θ, the posterior distribution is given by,

$$p_\theta(z|x) = \frac{p_\theta(x, z)}{p_\theta(x)}.$$

A Pyro model constitutes declaring

- z using `sample` primitive (essentially a function)
- x using `sample` with `obs` parameter set to the data, and
- θ using `param` primitive.

In addition, `plate` primitive allows for repetition in the model.

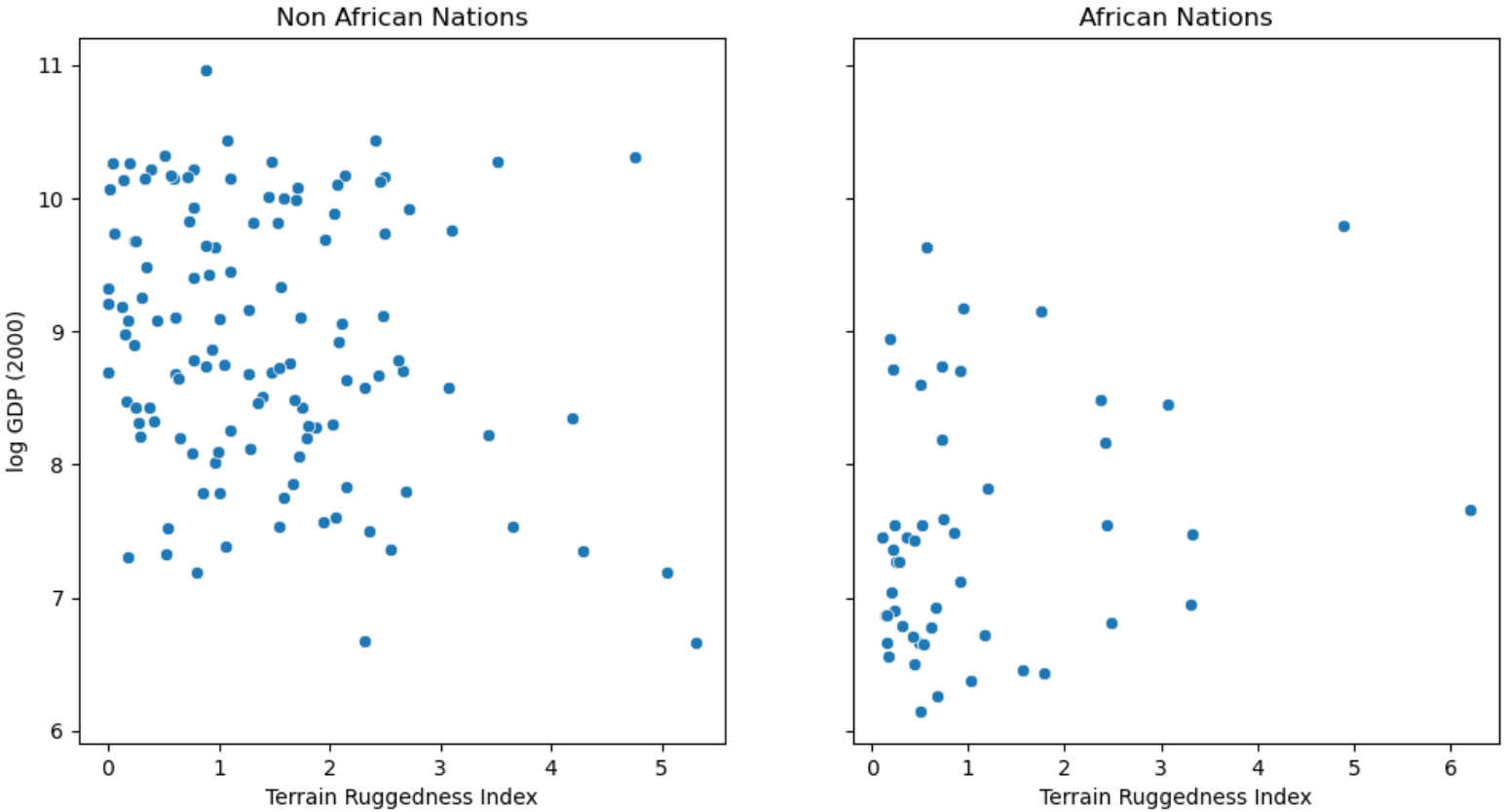
Pyro: Bayesian linear regression

Relationship between ruggedness of terrain to GDP.

- The ruggedness of terrain is inversely related to GDP, except in Africa.

$$GDP_i = a + b_a x_{i,\text{cont}} + b_r x_{i,\text{ruggedness}} + b_{ar} x_{i,\text{cont}} x_{i,\text{ruggedness}}.$$

Pyro: Bayesian linear regression



Pyro: Bayesian linear regression

Directed acyclic graphs depict joint distributions:

Circular nodes indicate random variables,

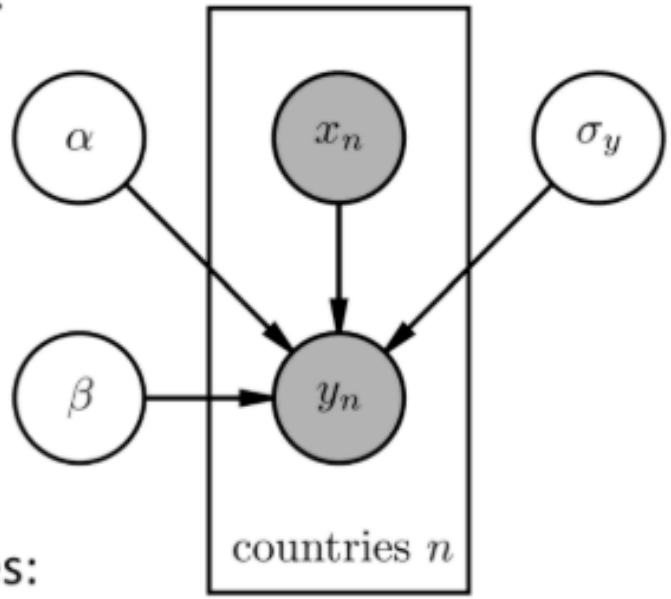
edges indicate dependence:

$$p(y_n \mid x_n, \alpha, \beta, \sigma)$$

Unshaded nodes are latent variables;

shaded nodes are observed variables.

"Plates" (rectangles) indicate independent copies:



$$p(y_1, \ldots, y_N \mid x_1, \ldots, x_N, \alpha, \beta, \sigma) = \prod_n p(y_n | x_n, \alpha, \beta, \sigma)$$

Figure 1: https://pyro.ai/examples/intro_long.html#Inference-in-Pyro

Pyro: Bayesian linear regression

```
import pyro.distributions as dist
import pyro.distributions.constraints as constraints

def simple_model(is_cont_africa, ruggedness, log_gdp=None):
    a = pyro.param("a", lambda: torch.randn(()))
    b_a = pyro.param("bA", lambda: torch.randn(()))
    b_r = pyro.param("bR", lambda: torch.randn(()))
    b_ar = pyro.param("bAR", lambda: torch.randn(()))
    sigma = pyro.param("sigma", lambda: torch.ones(()),
                       constraint=constraints.positive)

    mean = a + b_a * is_cont_africa + b_r * ruggedness +
           b_ar * is_cont_africa * ruggedness

    with pyro.plate("data", len(ruggedness)):
        return pyro.sample("obs", dist.Normal(mean, sigma),
                           obs=log_gdp)
```

Pyro: Bayesian linear regression

As it stands, the model is treating the regression coefficients as fixed parameters to be estimated.

To make it random, we need to turn `param` primitives to `sample` primitives.

Pyro: sample

```
def sample(
    name: str,
    fn: pyro.distributions.Distribution,
    *,
    obs: typing.Optional[torch.Tensor] = None,
    infer: typing.Optional[dict] = None
) -> torch.Tensor:
    ...
```

- `name`: Specify name of the random variable.
- `fn`: The distribution for `name`. Full list of distributions here.
- `obs`: If the variable is observed, set it by passing in `torch.Tensor` object.

Pyro: sample

```
import pyro.distributions as dist
import pyro.distributions.constraints as constraints

a = pyro.sample("a", dist.Normal(...))
b_a = pyro.sample("bA", dist.Normal(...))
b_r = pyro.sample("bR", dist.Normal(...))
b_ar = pyro.sample("bAR", dist.Normal(...))
sigma = pyro.sample("sigma", "???")

with pyro.plate("data", len(ruggedness)):
    return pyro.sample("obs", dist.Normal(mean, sigma),
                       obs=log_gdp)
```

Pyro: plate

Analogous to a `for` loop in probabilistic programming.

```
def plate(
    name: str,
    size: int,
    *,
    dim: Optional[int] = None,
    **other_kwargs
) -> contextlib.AbstractContextManager:
    ...
```

- `size`: number of items/data points in the plate.
- `subsample_size`: support mini-batch.

Pyro: plate

```
with pyro.plate("data", len(ruggedness)):
    return pyro.sample("obs", dist.Normal(mean, sigma),
                       obs=log_gdp)
```

is equivalent to,

```
result = torch.empty_like(ruggedness)
for i in range(len(ruggedness)):
    result[i] = pyro.sample(f"obs_{i}",
                dist.Normal(mean, sigma), obs=log_gdp[i])
return result
```

Pyro: `param`

A key-value parameter store (essentially a `dict`) that is persistent across model calls. In REPL (e.g., Jupyter-lab), advised to call `clear_param_store()` before re-running models/inference.

```
def param(
name: str,
init: Optional[Union[torch.Tensor, Callable[..., torch.Tensor]
        = None,
*,
constraint: torch.distributions.constraints.Constraint =
            constraints.real
) -> torch.Tensor:
    ...
```

- `init`: initial value as `torch.Tensor` or a function that returns `torch.Tensor`.
- `constraint`: the support set (real, positive, etc), see here.

Pyro: SVI

The `model` program specifies generative process for the data, specifying the relationship between latent variables, parameters, and the observed variables.

To perform variational approximation, we need to specify the variational approximation, which is referred to as the `guide` program.

- `guide` takes the same set of arguments as `model`.
- `guide` contains `param` and `sample` but without `obs` option (no observation).
- The variable names in `model` should appear in `guide` program; Pyro will match them 1-1.

Pyro: SVI

```
def model():
    pyro.sample("z_1", dist.Normal(...))
```

```
def guide():
    pyro.sample("z_1", dist.StudentT(...))
```

The names must match but the distributional specifications can differ.

Pyro: Bayesian linear regression

```
def custom_guide(is_cont_africa, ruggedness, log_gdp=None):
    a_loc = pyro.param('a_loc', lambda: torch.tensor(0.))
    a_scale = pyro.param('a_scale', lambda: torch.tensor(1.),
                         constraint=constraints.positive)
    sigma_loc = pyro.param('sigma_loc', lambda: torch.tensor(0.))
    weights_loc = pyro.param('weights_loc', lambda: torch.randn(3))
    weights_scale = pyro.param('weights_scale', lambda: torch.ones(3),
                               constraint=constraints.positive)
    a = pyro.sample("a", dist.Normal(a_loc, a_scale))
    b_a = pyro.sample("bA", dist.Normal(weights_loc[0], weights_scale[0]))
    b_r = pyro.sample("bR", dist.Normal(weights_loc[1], weights_scale[1]))
    b_ar = pyro.sample("bAR", dist.Normal(weights_loc[2], weights_scale[2]))
    sigma = pyro.sample("sigma", dist.LogNormal(sigma_loc, torch.tensor(0.05)))
    return {"a": a, "b_a": b_a, "b_r": b_r, "b_ar": b_ar, "sigma": sigma}
```

Pyro: inference

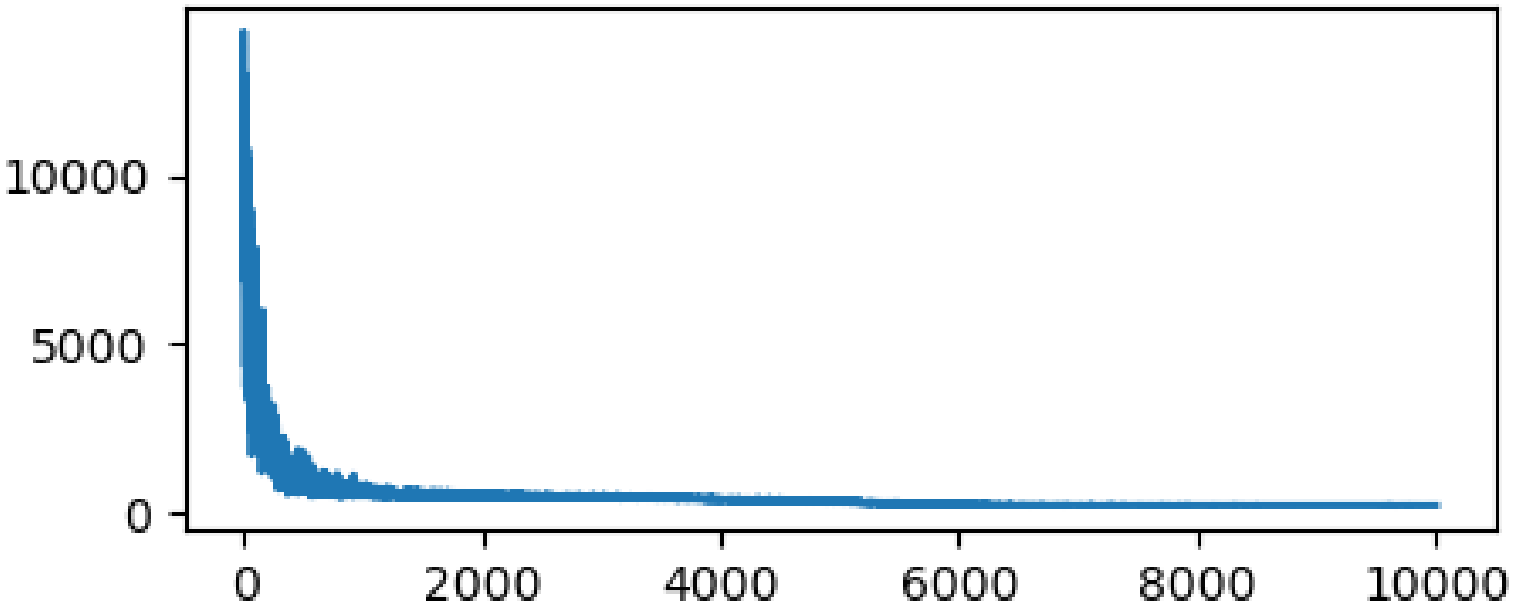
```
adam = pyro.optim.Adam({"lr": 0.02})
elbo = pyro.infer.Trace_ELBO()
svi = pyro.infer.SVI(model, auto_guide, adam, elbo)

losses = []
for step in range(10000 if not smoke_test else 2):
    loss = svi.step(is_cont_africa, ruggedness, log_gdp)
    losses.append(loss)
    if step % 100 == 0:
        logging.info("Elbo loss: {}".format(loss))
```

Pyro: inference

Trace ELBO accepts `num_particles` as an input:

```
pyro.infer.Trace_ELBO(
    num_particles=1,
    max_plate_nesting=inf,
    max_iarange_nesting=None,
    vectorize_particles=False,
    strict_enumeration_warning=True,
    ignore_jit_warnings=False,
    jit_options=None,
    retain_graph=None,
    tail_adaptive_beta=-1.0,
)
```

Pyro: inference



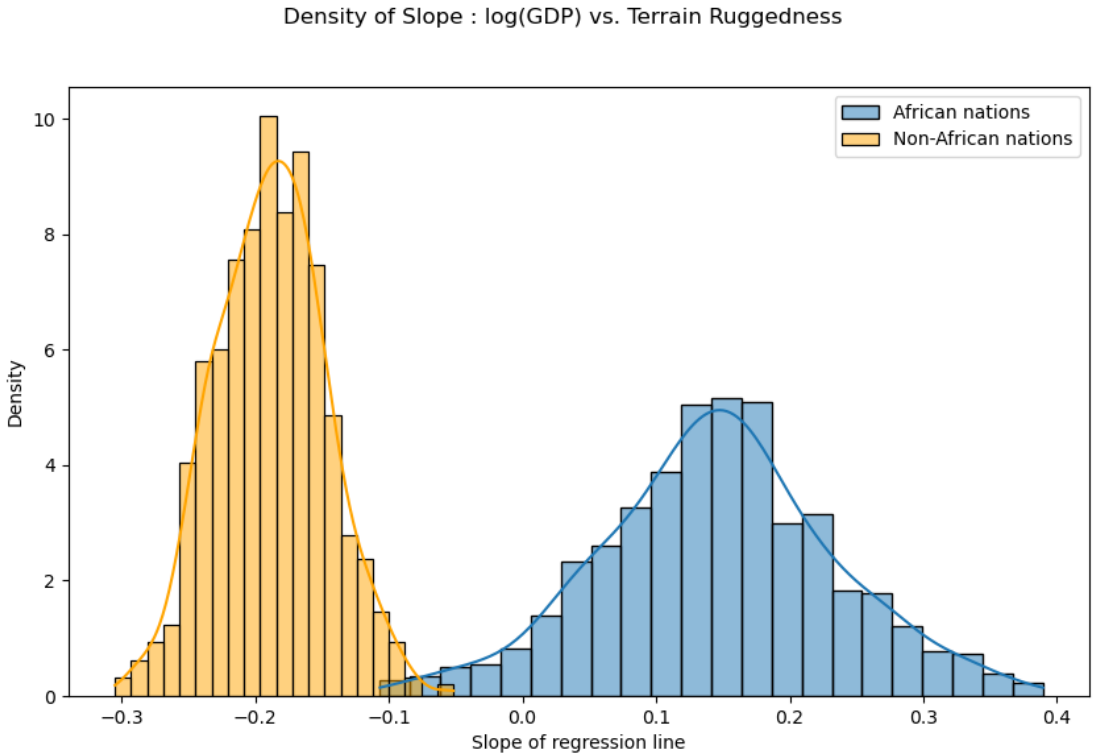
Pyro: inference

What can we do with the variational approximations?

Pyro: inference

What can we do with the variational approximations?

Generate samples $z_n \sim q_\psi(z|x_i)$ and plot the density.



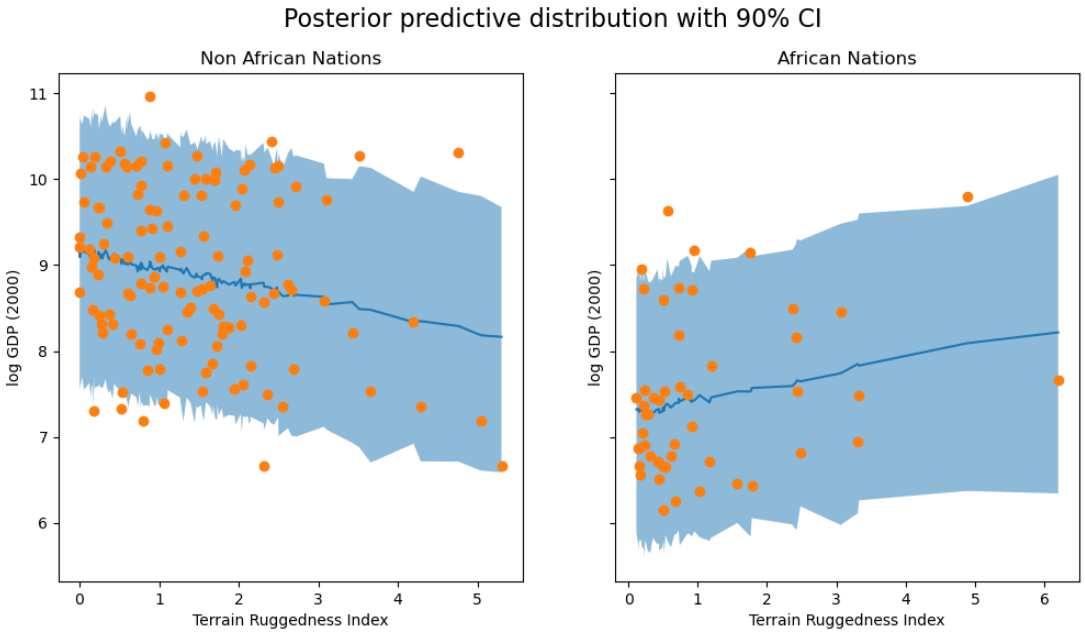
Pyro: inference

What can we do with the variational approximations?

Pyro: inference

What can we do with the variational approximations?

Generate samples $z_n \sim q_\psi(z)$ and construct predictive intervals for the observations:



Pyro: autoguide

Pyro provides automatic guides for standard problems.

Rather than specifying the `custom_guide` program, we can use

```
auto_guide = pyro.infer.autoguide.AutoNormal(model)
```

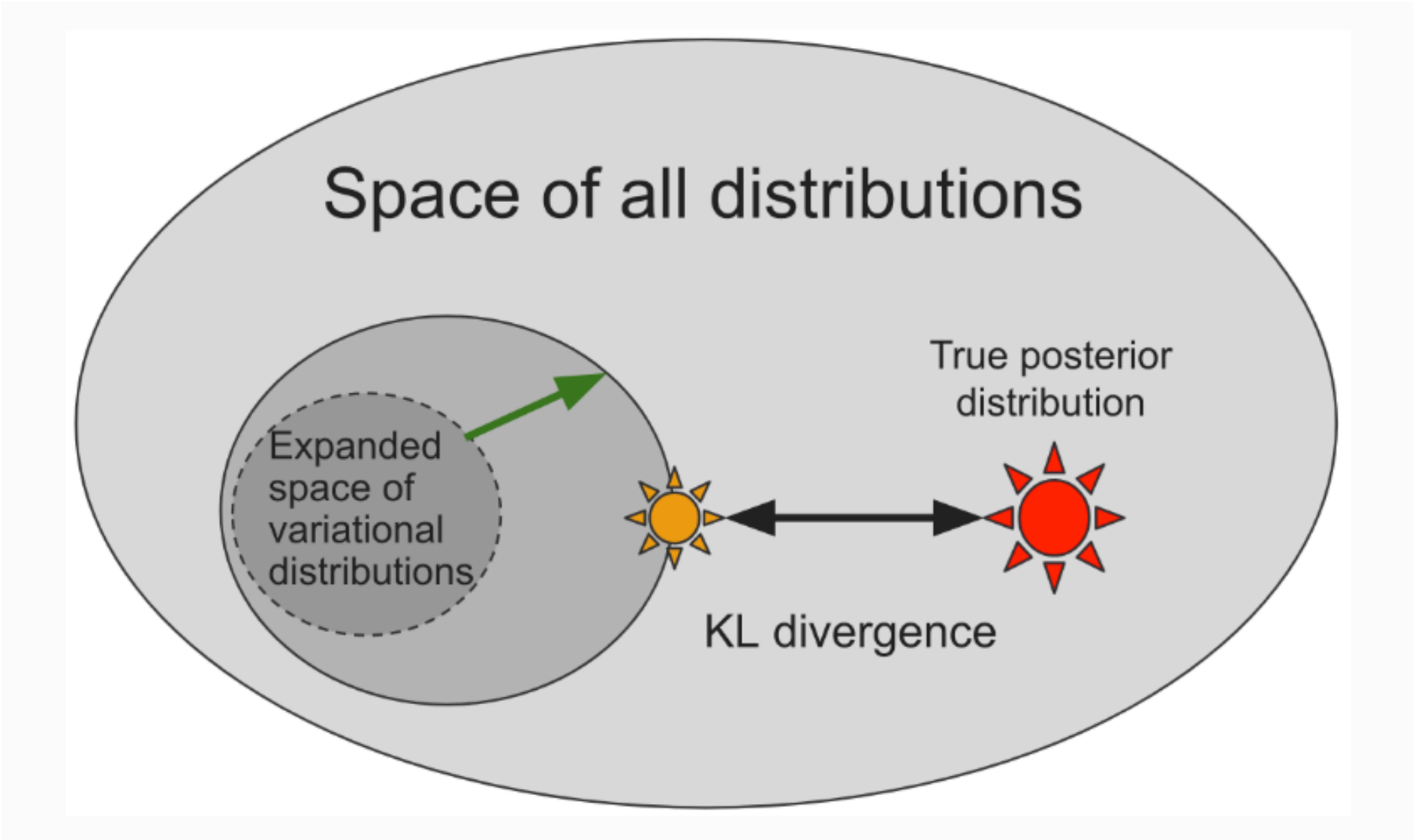
Pyro: autoguide

Full rank `guide`:

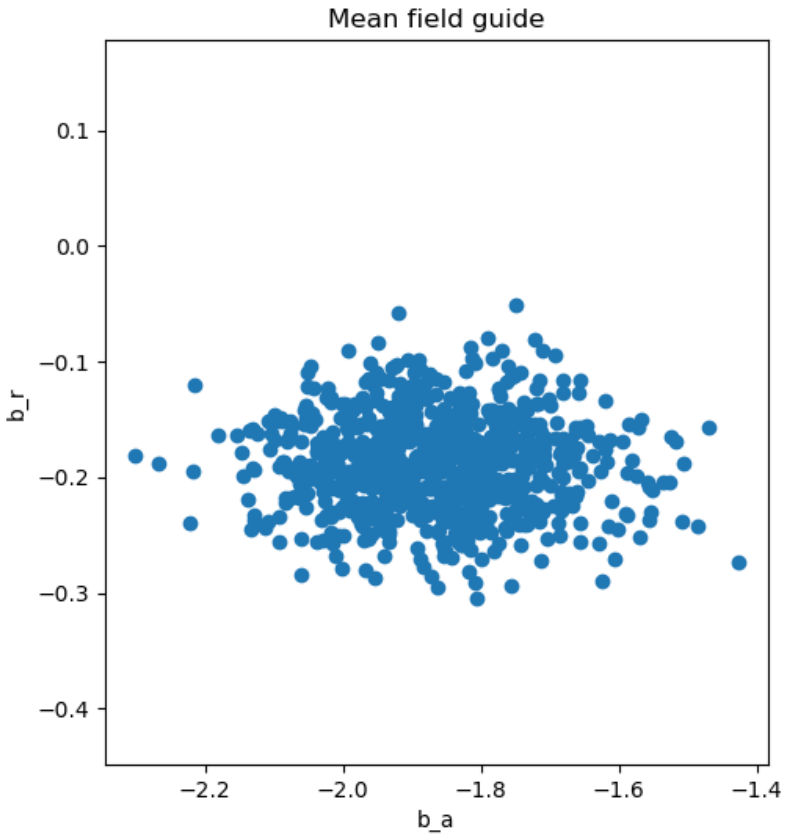
$$(a, b_a, b_r, b_{ar}, \log \sigma) \sim \mathsf{MVN}(\mu, \Sigma)$$

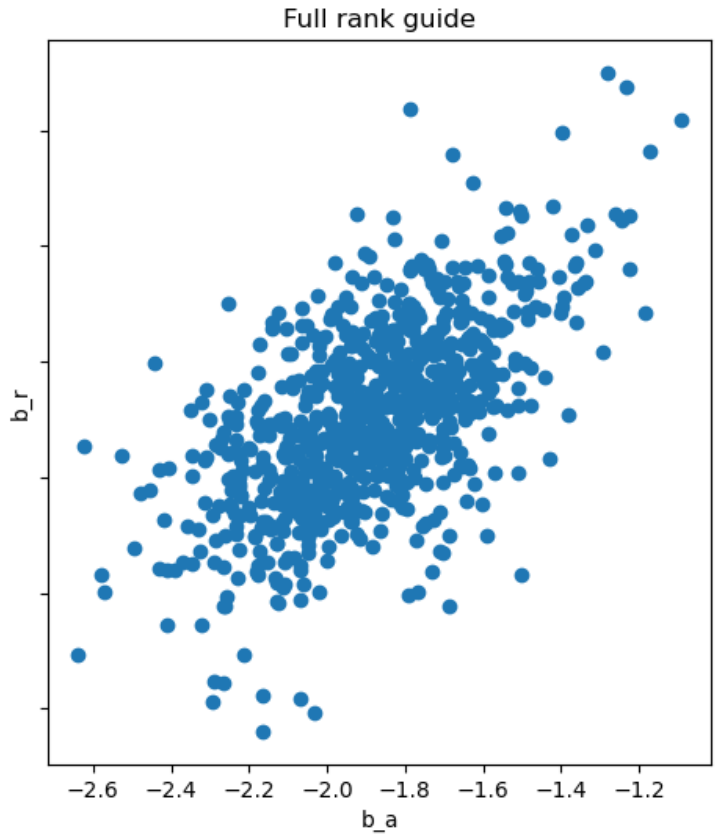
```
mvn_guide = pyro.infer.autoguide.AutoMultivariateNormal(model)
```

Pyro: autoguide

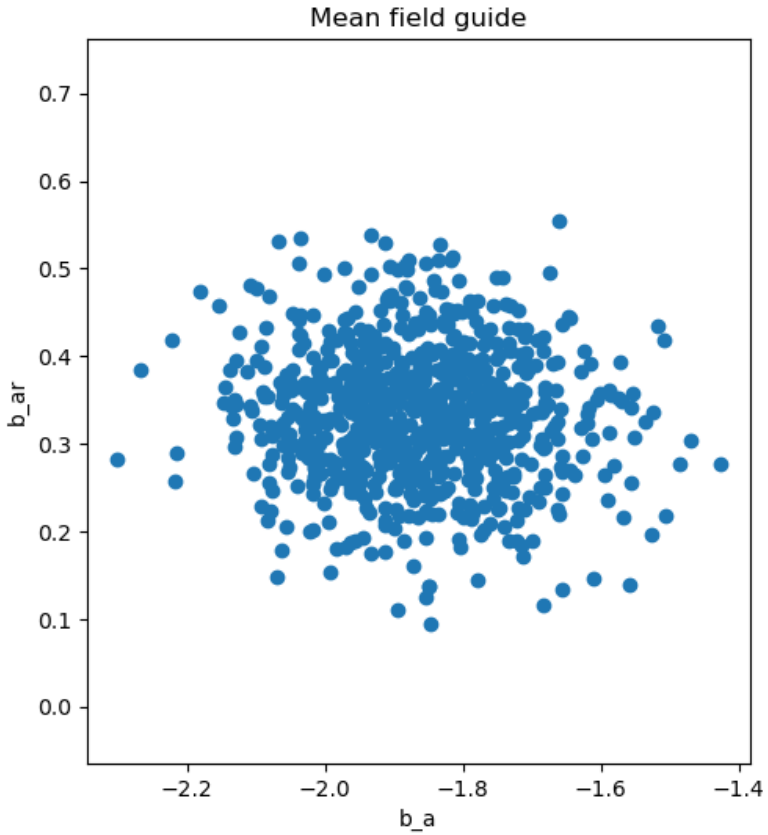


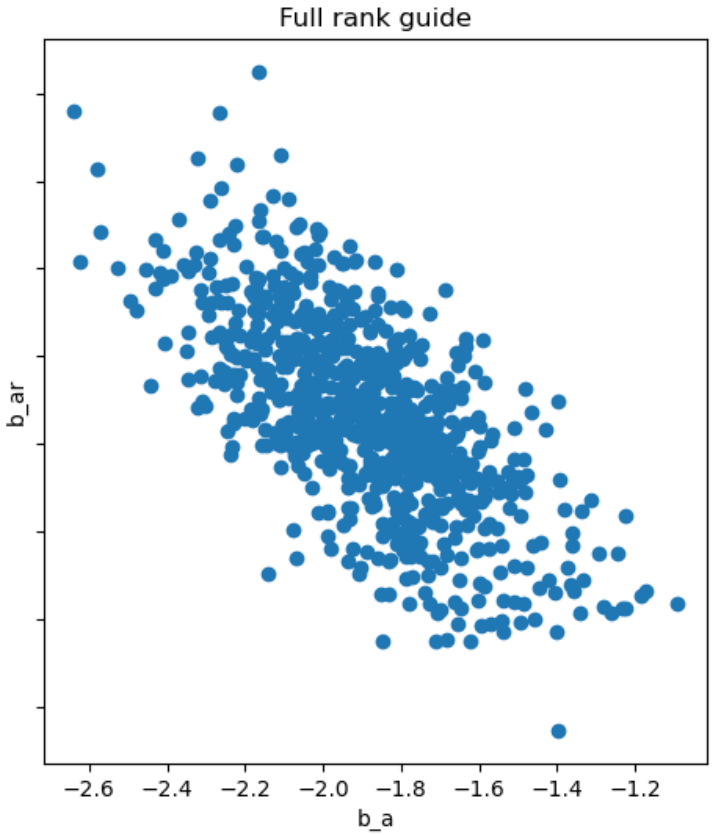
Pyro: autoguide



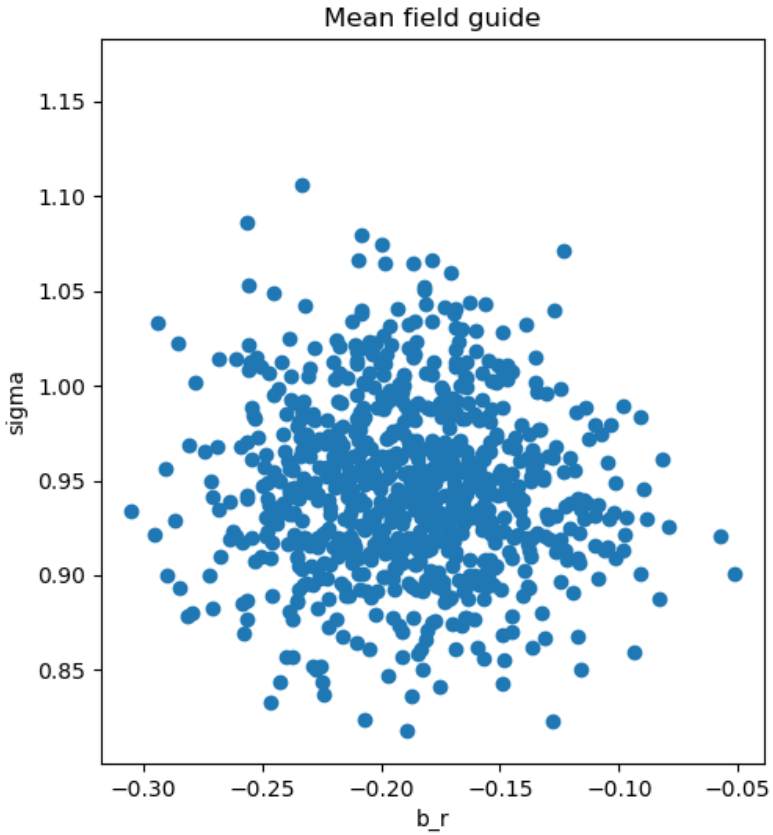


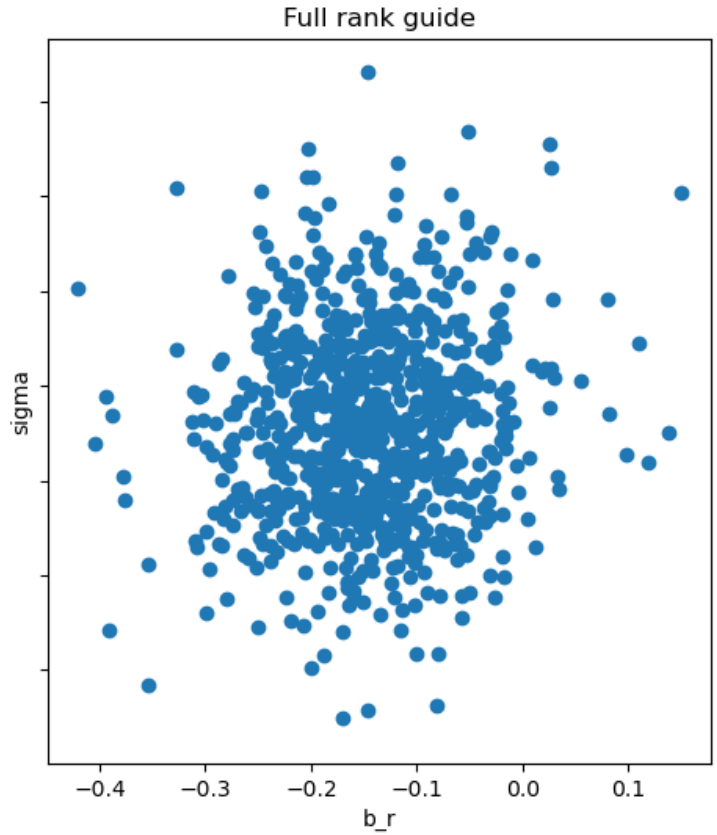
Pyro: autoguide



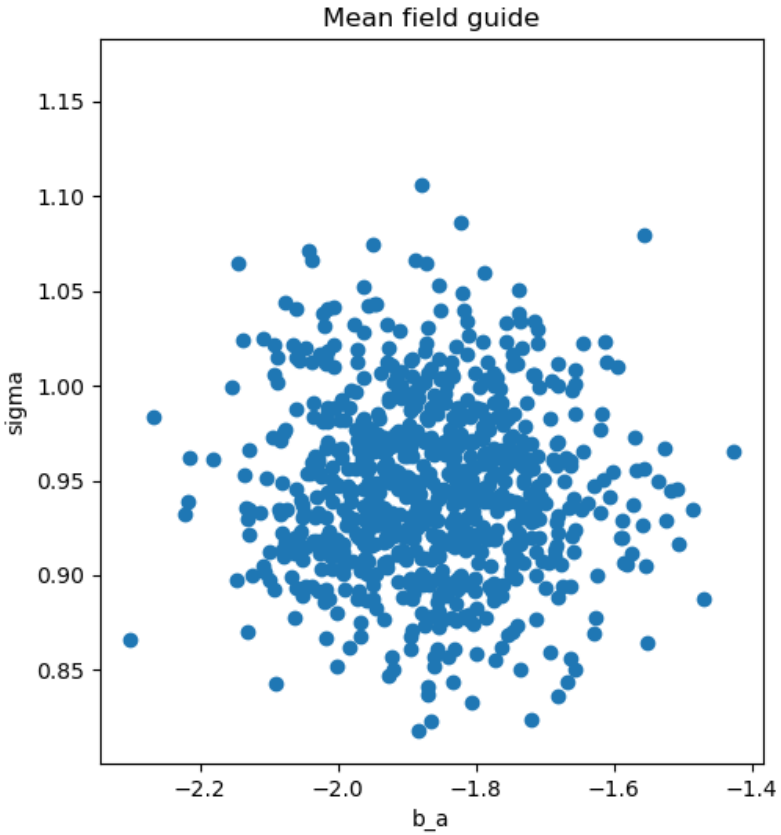


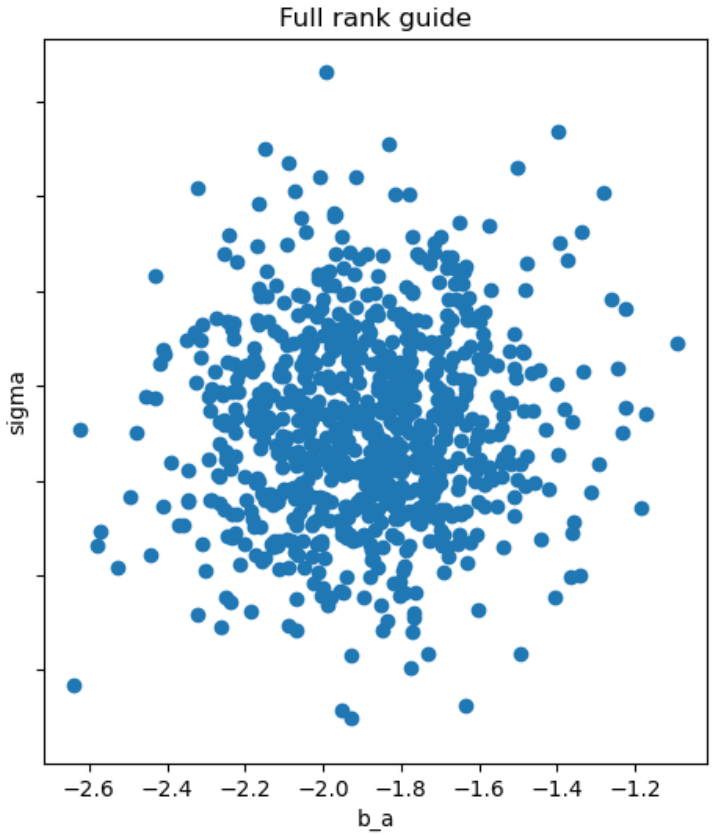
Pyro: autoguide



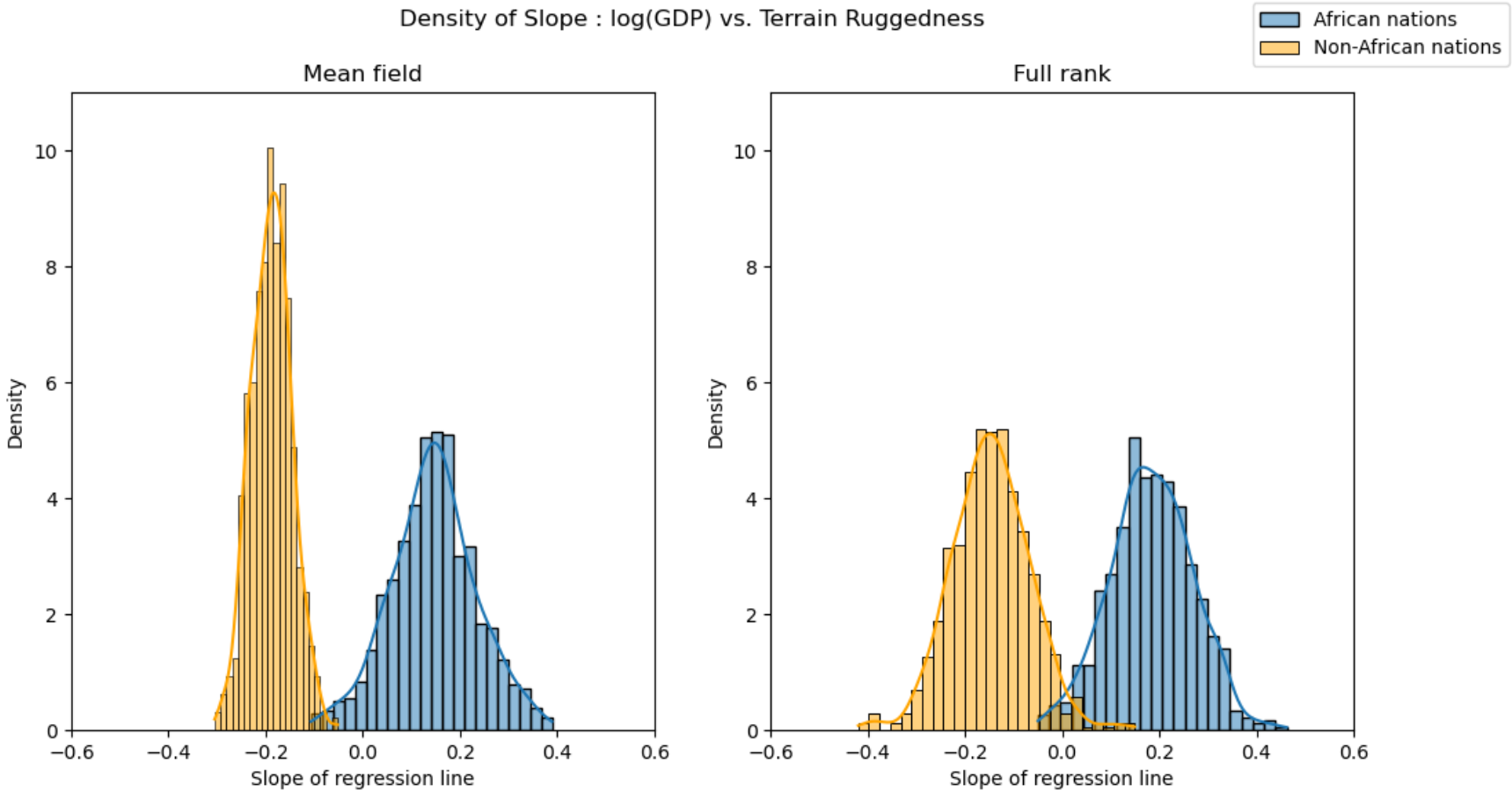


Pyro: autoguide





Pyro: autoguide



Mode seeking nature of KL

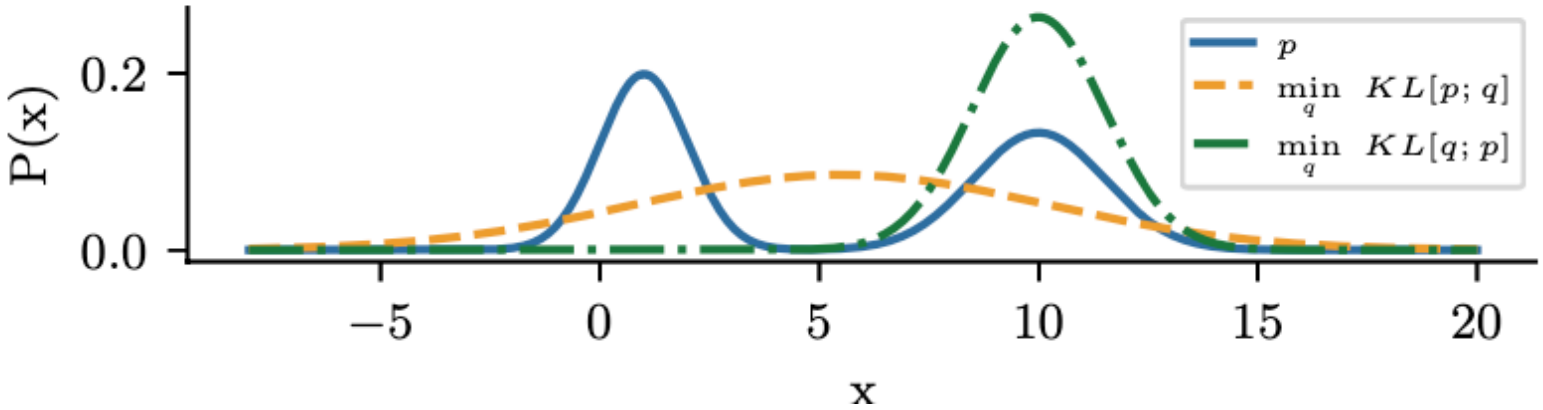


Figure 2: Figure 5.1 PML2

Importance Weighted Auto-Encoders (IWAE)

The recognition network $q_\psi(z|x)$ can be seen as an importance distribution for the target distribution $p_\theta(z|x)$.

Draw samples $z_n \sim q_\psi(z|x)$:

$$p_\theta(x) = \int p_\theta(x, z) dz \tag{1}$$

$$= \int \frac{p_\theta(x, z)}{q_\psi(z|x)} q_\psi(z|x) dz \tag{2}$$

$$\approx \frac{1}{N} \sum_{n=1}^{N} w(x, z_n), \tag{3}$$

where $w(x, z_n) = p_\theta(x, z_n)/q_\psi(z_n|x)$.

Importance Weighted Auto-Encoders (IWAE)

Let

$$\hat{p}_\theta(x) = \frac{1}{N}\sum_{n=1}^{N} w(x, z_n).$$

Using Jensen's inequality, we can show that IWAE provides multi-sample ELBO:

$$\begin{aligned}
\mathbb{E}_{z_{1:N}}\left[\log \hat{p}_\theta(x)\right] &\leq \log \mathbb{E}_{z_{1:N}}[\hat{p}_\theta(x)] && (4)\\
&= \log \frac{1}{N}\sum_{n=1}^{N} \int w(x, z_n) q_\psi(z_n|x) dz_n && (5)\\
&= \log \frac{1}{N}\sum_{n=1}^{N} \int p(x, z_n) dz_n && (6)\\
&= \log p(x). && (7)
\end{aligned}$$

Importance Weighted Auto-Encoders (IWAE)

IWAE provides tighter lower bound than simple averaging: let $z_n \sim q_\psi(z|x)$,

$$\log\left(\frac{1}{N}\sum_n w(x, z_n)\right) \geq \frac{1}{N}\sum_n \log w(x, z_n), \tag{8}$$

by Jensen since $\log$ is concave.

The RHS approximate ELBO,

$$\frac{1}{N}\sum_n \log w(x, z_n) = \frac{1}{N}\sum_n \log p_\theta(x, z_n) - \log q_\psi(z_n|x) \tag{9}$$

$$\approx \mathbb{E}_{z\sim q_\psi}[\log p_\theta(x, z_n) - \log q_\psi(z_n|x)]. \tag{10}$$

Importance Weighted Auto-Encoders (IWAE)

- IWAE objective reduces mass seeking behavior and leads to more diffuse variational approximation.
- Increasing the sample size makes the bound tighter but can also make the optimization problem more difficult.
- Empirical evidence shows that the sweet spot seems to be around 10 to 20 samples.
- IWAE seems to lead to better generalization/generative modeling but can also incur high variance in the weights.

Improvements

- Variational Inference for Monte Carlo Objective (VIMCO) – Mnih and Rezende (2016).
  - Proposes a variance-reduced gradient estimator for discrete multi-sample objectives.
  - Uses a leave-one-out approach to compute baselines for the importance weights, significantly reducing the variance.
- Multiple Importance Sampling ELBO (MISELBO) – Kviman et al (2022).
  - Aims to alleviate mode-seeking behavior by using a mixture of approximations (multiple proposals).
  - Combines them via importance sampling, thereby covering more of the posterior mass.