Simulation-Based Inference

Neural Estimation & BayesFlow

Šimon Kucharský

Neural Estimation

Recap

We have some data y and want to perform inference about model parameters θ.

Bayesian Inference

$$p(\theta \mid y) = \frac{p(\theta) \times p(y \mid \theta)}{p(y)},$$

where $p(y) = \int p(\theta) \times p(y \mid \theta) d\theta$.

$\rightarrow$ need to solve a difficult integral.

Options

- Analytic solution: Often times not possible.
- Numerical solution: Does not scale well.
- Brute force sampling: Does not scale well.
- MCMC sampling: Good option, but sometimes slow
- VI: Faster than MCMC, but less accurate

$\rightarrow$ All options require **evaluating** $p(y \mid \theta)$, $p(\theta)$!

- ABC: Simulation-based, but does not scale well, requires custom summary statistics...

Neural approximations

Use generative neural networks (coupling flows, flow matching, diffusion models, ...) to approximate distributions of interest.

Depending on which quantity we approximate:

- *Neural likelihood estimation*: Approximate $p(y \mid \theta)$
- *Neural posterior estimation*: Approximate $p(\theta \mid y)$

Also possible to approximate priors $p(\theta)$, marginal likelihoods $p(y)$, or likelihood ratios (NRE) $\frac{p(y|\theta)}{p(y)}$, ...

Two phases

1. Train the networks on *simulated* tuples (θ, y)
2. Use the networks on observed data y^{obs} for inference

Neural approximations

Use generative neural networks to approximate distributions of interest.
Train them on *simulated* tuples (θ, y).

Main advantages

- No need to *evaluate* $p(y \mid \theta)$, $p(\theta)$, only need to draw samples $(\theta^{(s)}, y^{(s)}) \sim p(\theta, y)$
- Often considerable (orders of magnitude) speed up during inference

Main disadvantages

- Less guarantees (e.g., how does the approximators perform when confronted with data not seen during training)
- Relies on extensive training (can be slow and expensive)

Neural Likelihood Estimation (NLE)

Approximate $p(y \mid \theta)$ with a generative neural network $q_\phi(y \mid \theta)$ (e.g., Normalizing flow), assuming we can *simulate* data from $p(y \mid \theta)$.

Use for inference

Substitute p with q_ϕ (*"surrogate likelihood"*) in traditional methods, e.g.,

- MLE: $\mathrm{argmax}_\theta \, q_\phi(y \mid \theta)$
- MAP: $\mathrm{argmax}_\theta \, q_\phi(y \mid \theta) \times p(\theta)$
- MCMC: Sample $\theta \propto q_\phi(y \mid \theta) \times p(\theta)$

NLE training

A normalizing flow (or another architecture that allows conditional density evaluation), $q_\phi(y \mid \theta)$

$$\mathsf{KL}\big(p(y \mid \theta) \mid\mid q_\phi(y \mid \theta)\big) = \mathbb{E}_{p(y|\theta)} \log \frac{p(y \mid \theta)}{q_\phi(y \mid \theta)}$$

With a proposal distribution $\tilde{p}(\theta)$, minimize the expected KL divergence,

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\, \mathbb{E}_{\tilde{p}(\theta)} \mathbb{E}_{p(y|\theta)} \big(\underbrace{\log p(y \mid \theta)}_{\text{constant wrt } \phi} - \log q_\phi(y \mid \theta) \big)$$

$$\hat{\phi} = \underset{\phi}{\operatorname{argmax}}\, \mathbb{E}_{\tilde{p}(\theta, y)} \big[\log q_\phi(y \mid \theta) \big]$$

NLE training

Monte Carlo approximation

$$\mathbb{E}_{\tilde{p}(\theta,y)}\big[\log q_\phi(y \mid \theta)\big] \approx \frac{1}{S}\sum_{s=1}^{S} \log q_\phi(y^{(s)} \mid \theta^{(s)})$$

with

$$\theta^{(s)} \sim \tilde{p}(\theta)$$
$$y^{(s)} \sim p(y \mid \theta^{(s)})$$

Neural Posterior Estimation (NPE)

Approximate directly $p(\theta \mid y)$ with a generative neural network $q_\phi(\theta \mid y)$ (e.g., Normalizing flow).

Use for inference

One trained, inference by

- sampling $\theta \sim q_\phi(\theta \mid y_{\text{obs}})$
- evaluating $q_\phi(\theta \mid y_{\text{obs}})$

  $\rightarrow$ a single pass through the network, no need for MCMC, optimization, ...

NPE training

A normalizing flow (or another architecture that allows conditional density evaluation), $q_\phi(\theta \mid y)$.

$$\mathsf{KL}\big(p(\theta \mid y) \mid\mid q_\phi(\theta \mid y)\big) = \mathbb{E}_{p(\theta|y)} \log \frac{p(\theta \mid y)}{q_\phi(\theta \mid y)}$$

With a prior distribution $p(\theta)$, minimize the expected KL divergence,

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\, \mathbb{E}_{p(\theta)} \mathbb{E}_{p(\theta|y)} \big(\underbrace{\log p(\theta \mid y)}_{\text{constant wrt } \phi} - \log q_\phi(\theta \mid y) \big)$$

$$\hat{\phi} = \underset{\phi}{\operatorname{argmax}}\, \mathbb{E}_{p(\theta,y)} \big[\log q_\phi(\theta \mid y) \big]$$

NPE training

Monte Carlo approximation

$$\mathbb{E}_{\tilde{p}(\theta,y)}\big[\log q_\phi(\theta \mid y)\big] \approx \frac{1}{S}\sum_{s=1}^{S} \log q_\phi(\theta^{(s)} \mid y^{(s)})$$

with

$$\begin{aligned} \theta^{(s)} &\sim p(\theta) \\ y^{(s)} &\sim p(y \mid \theta^{(s)}) \end{aligned}$$

Intuition

By symmetry, $p(\theta)p(y \mid \theta) = p(\theta, y) = p(y)p(\theta \mid y)$.

Training data $(\theta^{(s)}, y^{(s)})$ can be thought of as if

$$\begin{aligned} y^{(s)} &\sim p(y) \\ \theta^{(s)} &\sim p(\theta \mid y^{(s)}) \end{aligned}$$

Samples of $\theta^{(s)}$ conditioned on $y^{(s)}$ can be thought of as sampled from $p(\theta \mid y^{(s)})$. Therefore, learning the distribution of $\theta^{(s)}$ conditioned on $y^{(s)}$ gives us the approximation of the target posterior.

Summary Statistics

Datasets $y = y_1, \dots, y_n$ of varying sizes n, but GNNs typically needs fixed sized inputs

Probability symmetries - e.g., for exchangeable data we would like $q_\phi(\theta \mid y_1, \dots, y_n) = q_\phi(\theta \mid \pi(y_1, \dots, y_n))$

Use data summary h such that $p(\theta \mid y) = p(\theta \mid h(y))$ (i.e., the summary h is *sufficient* wrt model parameters θ).

Often, we do not know how to calculate sufficient statistics or they may not exist.

Summary Networks

Use a neural network h_ψ that returns data embeddings of fixed size.

Reflect probabilistic symmetries in the data, e.g.,

- DeepSet h_ψ to summarise exchangeable data (sets), $h_\psi(y_1, \dots, y_n) = h_\psi(\pi(y_1, \dots, y_n))$
- LSTM/Transformers for ordered time series
- CNNs for images

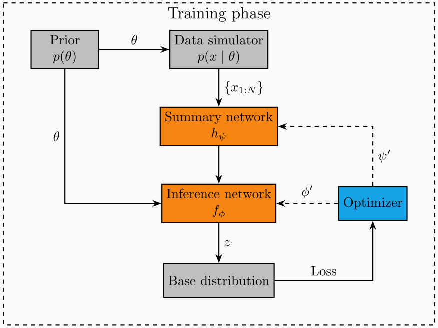
Train the posterior network and the summary network jointly,

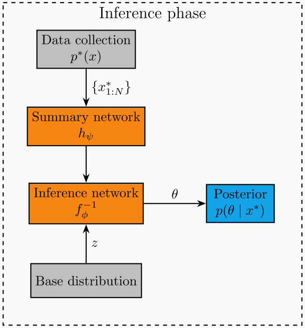
$$\hat{\phi}, \hat{\psi} = \underset{\phi,\psi}{\operatorname{argmax}}\, \mathbb{E}_{p(\theta,y)}\big[\log q_\phi(\theta \mid h_\psi(y))\big]$$

$\rightarrow$ *approximately* sufficient statistics, $p(\theta \mid y) \approx q_\phi(\theta \mid h_\psi(y))$.

Amortized Bayesian Inference (ABI)

Amortization: Pay the cost of inference upfront, reap the benefits later





Amortized Bayesian Inference (ABI)

Amortization: Pay the cost of inference upfront, reap the benefits later

Training

- Simulate data and parameters
- Train neural networks
- Slow, resource consuming process

Inference

- Observed data
- Obtain samples of θ from q_ϕ
- Fast, cheap process
- Typically milliseconds to seconds, *orders of magnitude faster than MCMC*

Uses of ABI

- Fit model to many *synthetic* datasets (network/model diagnostics)
- Fit model to many empirical datasets
- Fit model many times to a single dataset (e.g., loo)
- Online inference (refit the model as the data come in)
- Likelihoods (or priors) not analytically tractable

Simulation gaps

The approximation $q_\phi(\theta \mid h_\psi(y)) \approx p(\theta \mid y)$ holds if y^{obs} is similar to training data.

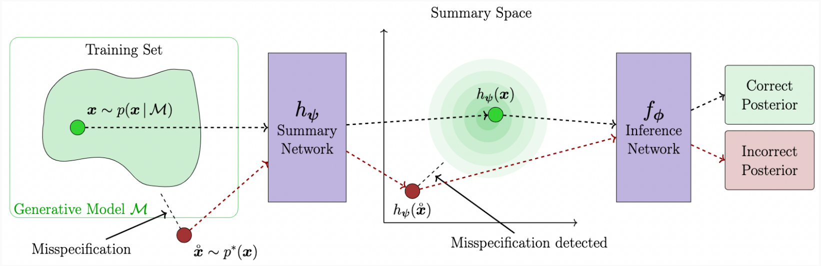
If y^{obs} is atypical (under the statistical model $p(\theta, y)$), inference can be *arbitrarily unreliable*.

Big topic in current research.

Example remedies:

1. Check if y^{obs} lies in the typical set. If not, **do not trust** inference.
2. Post-hoc correction (e.g., Pareto-smoothed importance sampling) - requires analytic likelihoods, only works for relatively minor deviations.
3. Include empirical data during training.

Check typicality of data



Summary Space
Training Set
$\boldsymbol{x} \sim p(\boldsymbol{x} \mid \mathcal{M})$
Generative Model $\mathcal{M}$
Misspecification
$\mathring{\boldsymbol{x}} \sim p^*(\boldsymbol{x})$
h_ψ
Summary Network
$h_\psi(\boldsymbol{x})$
$h_\psi(\mathring{\boldsymbol{x}})$
Misspecification detected
f_ϕ
Inference Network
Correct Posterior
Incorrect Posterior

Self-consistency

$$p(\theta \mid y) = \frac{p(\theta)p(y \mid \theta)}{p(y)} \longleftrightarrow p(y) = \frac{p(\theta)p(y \mid \theta)}{p(\theta \mid y)}$$

Notice that $p(y)$ is a *constant* wrt θ: it does not matter which values we plug in the RHS.

Additional loss:

$$\mathcal{L}_{\mathsf{SC}} = \mathsf{Var}_{\theta \sim q(\theta \mid y^{\mathsf{obs}})} \log \frac{p(\theta)p(y^{\mathsf{obs}} \mid \theta)}{q(\theta \mid y^{\mathsf{obs}})}$$

Evaluated on empirical data makes sure that $q(\theta \mid y^{\mathsf{obs}})$ is *self-consistent* (satisfies the Bayes theorem equality above).

Could be also used with non-analytic likelihoods, in that case we replace $p(y^{\mathsf{obs}} \mid \theta)$ with $q(y^{\mathsf{obs}} \mid \theta)$.

Sequential Neural Estimation

Amortization is powerful because it allows fast inference for many datasets.

But, what if we *only* need inference for a single dataset y^{obs}?

- Training on the whole region $\theta, y \sim p(\theta, y)$ might be wasteful (training data far away from y^{obs})
- Ideally, we would like to simulate data as close as possible to y^{obs}

Sequential Neural Estimation

Use a proposal distribution $\tilde{p}(\theta)$ such that the simulated data $y^{(s)} \sim p(y \mid \theta^{(s)})$ is close to the observed data.

The simulated data will be close to the empirical data if the parameters used for simulation are plausible under the empirical data.

Iteratively (sequentially) refine the proposal distribution such that we end up with $\tilde{p}(\theta) \approx p(\theta \mid y^{\text{obs}})$.

Sequential Neural Likelihood Estimation (SNLE)

Algorithm 1: SNLE

```
Data: Data $y^{\text{obs}}$
Result: $\tilde{p}_R(\theta) \approx p(\theta \mid y^{\text{obs}})$
1  Initialize training data $D = \{\}$, likelihood approximator $q_\phi(y \mid \theta)$;
2  Set $\tilde{p}_0(\theta) = p(\theta)$;
3  for r in 1:R do
4      for s in 1:S do
5          sample $\theta^{(s)} \sim \tilde{p}_{r-1}(\theta)$ with MCMC;
6          simulate $y^{(s)} \sim p(y \mid \theta^{(s)})$;
7          add $(\theta^{(s)}, y^{(s)})$ to $D$;
8      end
9      re-train $q_\phi(y \mid \theta)$ on $D$;
10     $\tilde{p}_r(\theta) \propto q_\phi(y^{\text{obs}} \mid \theta) \times p(\theta)$
11 end
```

Sequential Neural Posterior Estimation (SNPE)

Same idea as SNLE: Iteratively refine the proposal $\tilde{p}(\theta)$ to be close to $p(\theta \mid y^{\text{obs}})$.

Challenge: Training on samples generated from $\tilde{p}(\theta) \times p(y \mid \theta)$ leads to a posterior approximation $\tilde{q}(\theta \mid y) \propto \tilde{p}(\theta) \times p(y \mid \theta)$.

Need to apply corrections to obtain $q(\theta \mid y) \propto p(\theta) \times p(y \mid \theta) = \tilde{q}(\theta \mid y)\frac{p(\theta)}{\tilde{p}(\theta)} \to$ different options:

- SNPE-A: Post-hoc correction
- SNPE-B: Importance weights during training
- SNPE-C: Direct training of the corrected posterior

Summary

Generative neural networks can approximate complex distributions such as likelihoods (NLE), posteriors (NPE), etc. These models are trained on simulated data $\rightarrow$ no need for analytic likelihoods.

We must be careful about inferences when the data is far away from that the networks saw during training $\rightarrow$ Less guarantees than likelihood-based methods!

We split the workflow into two phases:

1. Training (compute-intensive)
2. Inference (fast, reusable) $\rightarrow$ *amortized* Bayesian inference

Sequential methods focus training on regions relevant to the empirical data, improving efficiency during training but tying inference to a specific dataset (removing *amortization*).

BayesFlow

Software

Reliance of machine learning $\rightarrow$ dominance of Python

- `BayesFlow`:
  - Main focus on ABI, other approaches possible
  - Developed (in part) in our lab
- `sbi`:
  - Amortized and Sequential approaches

In the rest of the course we will focus on using `BayesFlow` for ABI.

BayesFlow

- Python library built on `keras`
  - Use `tensorflow`, `PyTorch`, or `JAX` as a backend
- User friendly, modular, scalable
- Provides helper functionality facilitating the entire pipeline (more later)

Resources

- `BayesFlow` website: documentation, examples
- `BayesFlow` forum: ask questions about your use cases
- `BayesFlow` repository: contribute, submit bug reports, feature requests

Uses of BayesFlow

Generative modeling tasks:

- NPE: $p(\theta \mid y)$
- NLE: $p(y \mid \theta)$
- Amortized Model Comparison: $p(\mathcal{M} \mid y)$
- Amortized Point Estimation: $\widehat{\theta} \mid y$

Sequential methods not (yet implemented)

We will focus on NPE

NPE with BayesFlow

$$p(\theta \mid y) \approx q_\phi(\theta \mid h_\psi(y))$$

Pipeline

1. Statistical model definition (through a *simulator*)
2. Approximator definition (*neural networks*, *adapter*)
3. Network training
4. Network diagnostics
5. Inference

Pipeline

1. **Statistical model definition (through a *simulator*)**
2. Approximator definition (*neural networks*, *adapter*)
3. Network training
4. Network diagnostics
5. Inference

Model definition – simulator

- Define the joint model $p(\theta, y)$ with a simulator
- Object with a `.sample` method, `bf` expects *batched output*

```
class Simulator(bf.simulators.Simulator):
    def sample(self, batch_size):
        theta = np.random.beta(a=1, b=1, size=batch_size)
        x = np.random.binomial(n=10, p=theta)
        return dict(theta=theta, x=x)

simulator = Simulator()
dataset = simulator.sample(100)
```

Helper to define simulators

`bf.make_simulator` simplifies the model definition

- break complex models into simpler parts
- auto-batch simulations

```python
def prior():
    return dict(theta = np.random.beta(a=1, b=1))

def likelihood(theta):
    return dict(x = np.random.binomial(n=10, p=theta))

simulator = bf.make_simulator([prior, likelihood])
dataset = simulator.sample(10)
```

Data of varying dimensions

Within a batch, data must be of fixed shape.

But statistical models do not always produce fixed sized outputs.

Strategies

- Padd arrays to the same shape
- Vary shape *between* batches

Pad arrays

```
def context():
    return dict(n = np.random.randint(10, 101))

def prior():
    return dict(mu = np.random.normal(0, 1))

def likelihood(mu, n):
    x = np.zeros(101)
    x[:n] = np.random.normal(mu, 1, size=n)

    observed = np.zeros(0)
    observed[:n] = 1

    return dict(x=x, observed=observed)

simulator = bf.make_simulator([context, prior, likelihood])
```

Vary shape *between* batches

Constant "sample size" within a batch

- `meta_fn`: function that runs *once* per simulation

```python
def context(batch_size):
    return dict(n=np.random.randint(10, 101))

def prior():
    return dict(mu=np.random.normal(0, 1))

def likelihood(mu, n):
    return dict(x=np.random.normal(mu, 1, size=n))

simulator = bf.make_simulator(
  [prior, likelihood], meta_fn=context)
```

Padding vs batched context

Padding

- More verbose
- More general
- Works well with any setup

Batched context

- Cleaner code
- Limited in use
  - e.g., cannot effectively use in offline training (more later)

Pipeline

1. Statistical model definition (through a *simulator*)
2. **Approximator definition (*neural networks*, *adapter*)**
3. Network training
4. Network diagnostics
5. Inference

Approximator definition

Components

1. **Inference network**
   - Generate posterior given (summary of) data
2. **Summary network** (optional)
   - Return fixed size data embeddings (data summary)
3. **Adapter**
   - Configure simulation outputs to be passed into the networks

Inference network

Generate a posterior distribution of parameters conditionally on the data

A generative neural network

Differenct choices of architectures, e.g., Coupling flows (affine, spline), FlowMatching,...

```
inference_network = bf.networks.CouplingFlow()
```

```
inference_network = bf.networks.FlowMatching()
```

What architecture to pick?

Coupling flow

- Slow training
- Less expressive
- Faster during inference

Flow matching

- Fast training
- More expressive
- Slower inference on CPU

- **Affine coupling** for low-dimensional, uni-modal posteriors
- **Spline coupling** for harder posteriors, including multi-modal
- **Flow matching** for highly complex posteriors
  - or when inference speed is not of concern
  - or if GPUs available

Summary network

Fixed-sized embeddings of (possibly) variable-sized data

Makes the job easier for the inference network

Approximately sufficient statistics

$$p(\theta \mid y) \approx q_\phi(\theta \mid h_\psi(y))$$

What architecture to pick?

Reflect probabilistic symmetries in the data

- Exchangeable data: Deep Sets, Set Transformers
- Time series data: RNN, LSTM, Transformer
- Images: CNN

Examples

```
summary_network = bf.network.DeepSet()
```

```
summary_network = bf.networks.LSTNet()
```

Adapter

Configures inputs to be passed into network (training, inference)

Key ouputs

$$q(\theta \mid x, h(y))$$

- `inference_variables`
  - variables whose distribution we learn during training and predict during inference
- `inference_conditions`:
  - "direct" conditions that are passed directly into the inference net
- `summary_variables`:
  - variables passed into a summary net h whose output is passed into the inference net

Adapter example

```
adapter = (bf.Adapter()
  .concatenate(["mu", "sigma"], into="inference_variables")
  .rename(["n"], to="inference_conditions")
  .concatenate(["x", "observed"], into="summary_variables")
)

data = simulator.sample(10)
adapter(data)
```

- Transform variables
  - `.standardize`,`.sqrt`, `.log`, `.constrain`
- Reshape variables
  - `.as_set`, `.as_time_series`, `.broadcast`, `.one_hot`
- … and many more operations

Approximator

An interface for posterior approximation

Collects the components needed for *inference*

```
approximator = bf.approximators.ContinuousApproximator(
    inference_network=inference_network,
    summary_network=summary_network
    adapter=adapter
)
```

Workflow

An alternative for an `approximator`

Collects components used for *training*, *inference*, and *diagnostics*

```
workflow = bf.BasicWorkflow(
    inference_network=inference_network,
    summary_network=summary_network
    adapter=adapter,
    simulator=simulator
)
```

Pipeline

1. Statistical model definition (through a *simulator*)
2. Approximator definition (*neural networks*, *adapter*)
3. **Network training**
4. Network diagnostics
5. Inference

Network training

Similar to training any other model in `keras`

- Optimizer (e.g. `keras.optimizers.Adam`)
- Training(/simulation) budget and regime
  - Number of epochs, number of batches, batch size
  - Online training: Generate data on the fly (for each epoch)
  - Offline training: Train on presimulated data (each epoch same data)
- Compile and train via `approximator.fit`

`BasicWorkflow` makes this easier: Predefined, reasonable settings

Pipeline

1. Statistical model definition (through a *simulator*)
2. Approximator definition (*neural networks*, *adapter*)
3. Network training
4. **Network diagnostics**
5. Inference

Diagnostics

The ultimate goal is to determine whether

$$p(\theta \mid y) \approx q_\phi(\theta \mid h_\psi(y))$$

holds.

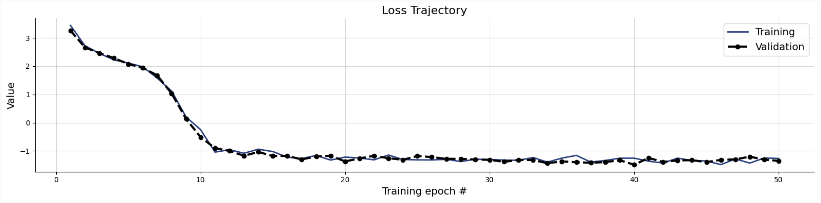
Helper functions in the `bf.diagnostics` module.

Networks convergence

`bf.diagnostics.plots.loss`

Assess the training/validation loss for convergence.

Generally for DL, not specific to ABI.



Approximator diagnostics

$$\theta^* \underbrace{\text{estimated by } p(\theta \mid y)}_{\text{Model sensitivity}} \overbrace{\text{approximated by } q_\phi(\theta \mid h_\psi(y))}^{\text{Computational faithfulness}}$$

1. "Computational faithfulness' ': How close is the approximation to the *true* posterior?
   - Simulation-based calibration
2. "Model sensitivity' ': How much can be expect to learn from the data?
   - Parameter recovery, posterior contraction, posterior z-score

Approximator diagnostics – procedure

1. Draw test data from the simulator (S fresh simulations)
2. For each dataset, draw M posterior samples from the approximator

```
test_data = simulator.sample(1000)
prior = dict(mu=test_data["mu"], sigma=test_data["sigma"])
posterior = approximator.sample(
  num_samples=500,
  conditions=test_data)
```

Use the test data + posterior samples for validation.

Big advantage of ABI!

Simulation-based calibration (SBC)

Check computational faithfullness of the approximator. See slides from week 9.

Simulate from the joint model

$$p(\theta)p(y \mid \theta) = p(\theta, y) = p(y)p(\theta \mid y)$$

$$\theta^{(s)} \sim p(\theta) \qquad y^{(s)} \sim p(y)$$

$$y^{(s)} \sim p(y \mid \theta^{(s)}) \qquad \theta^{(s)} \sim p(\theta \mid y^{(s)})$$

Obtain samples from the approximator

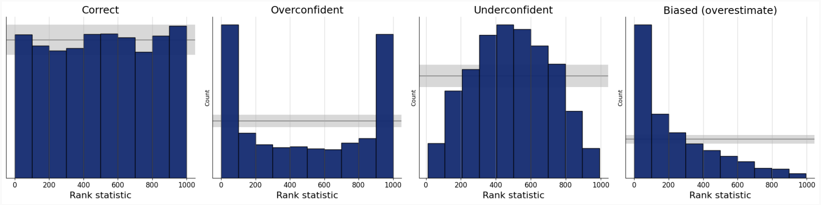
$$\theta^{(s,1)}, \dots, \theta^{(s,M)} \sim q_\phi(\theta \mid h_\psi(y^{(s)}))$$

If $p(\theta \mid y^{(s)}) \approx q_\phi(\theta \mid h_\psi(y^{(s)}))$

$$r^{(s)} = \mathsf{rank}(\theta^{(s)}, (\theta^{(s,1)}, \dots, \theta^{(s,M)})) \sim \mathsf{Categorical}\left(\frac{1}{M+1}\right)$$

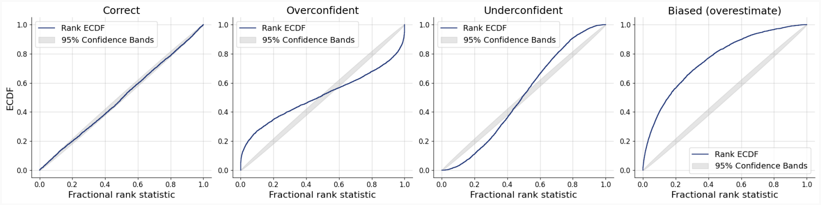
SBC Histograms

```
fig=bf.diagnostics.calibration_histogram(
    estimates=posterior,
    targets=prior)
```



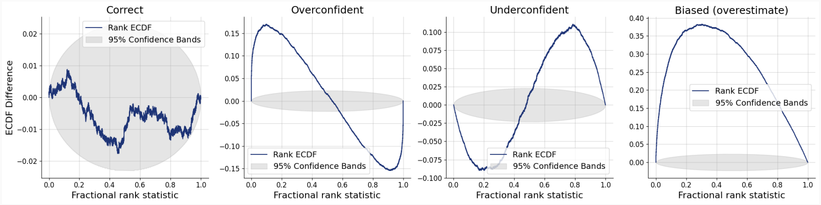
SBC ECDF

```
fig=bf.diagnostics.calibration_ecdf(
    estimates=posterior,
    targets=prior)
```



SBC ECDF Difference

```
fig=bf.diagnostics.calibration_ecdf(
    estimates=posterior,
    targets=prior,
    difference=True)
```



SBC Interpretation

- All faithful approximators pass the SBC check
- Some unfaithful approximators may pass the SBC check
- Simulation budget (how many datasets S)
- Approximation precision (how many posterior draws M)

$\rightarrow$ degree/types of miscalibration rather than binary decision

If SBC check fails

Indicates a problem in at least one part of the pipeline:

1. **Simulator**: Check for coding errors,...
2. **Statistical model**: Change the model, reparametrize,...
3. **Approximator**: Check for coding errors (is adapter correct?), change specifications of inference/summary networks,...

Model sensitivity

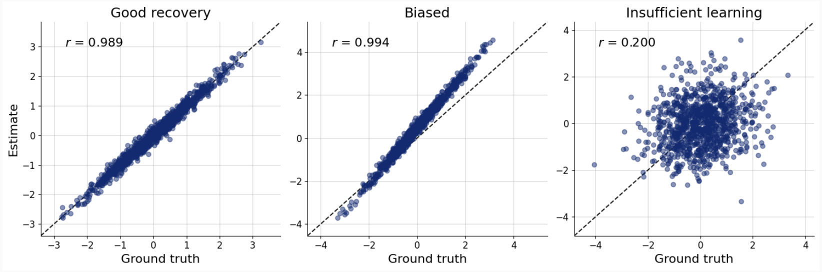
How much can we expect to learn from the data?

1. Can we recover the *true* parameters
2. How close is the posterior mean to the *true* parameters, relatively speaking
3. How much uncertainty is removed after seeing the data

Parameter recovery

- Plot the true parameter values $\theta^{(s)}$ against a posterior point estimate (mean, median)

```
fig=bf.diagnostics.plots.recovery(
  estimates=posterior,
  targets=prior)
```



Posterior z-score and posterior contraction

- How well does the estimated posterior mean match the true parameter used for simulating the data?

$$z = \frac{\mu_{\text{post}} - \theta}{\sigma_{\text{post}}}$$

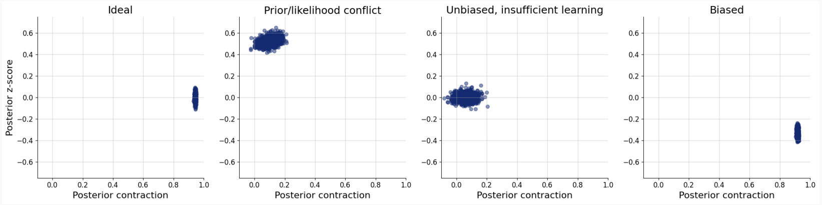
- How much uncertainty is removed after updating the prior to posterior?

$$\text{contraction} = 1 - \frac{\sigma^2_{\text{post}}}{\sigma^2_{\text{prior}}}$$

(means and variances are estimated from the posterior samples)

Z-score vs. contraction plot

```
fig=bf.diagnostics.plots.z_score_contraction(
    estimates=posterior,
    targets=prior)
```



What if I get poor results

Can indicate problems with the approximator (see SBC)

Can indicate issues with the statistical model/data

- Use more informative priors
- Change the model
- Add more data (sample size, additional variables)

Pipeline

1. Statistical model definition (through a *simulator*)
2. Approximator definition (*neural networks*, *adapter*)
3. Network training
4. Network diagnostics
5. **Inference**

Obtain posterior samples

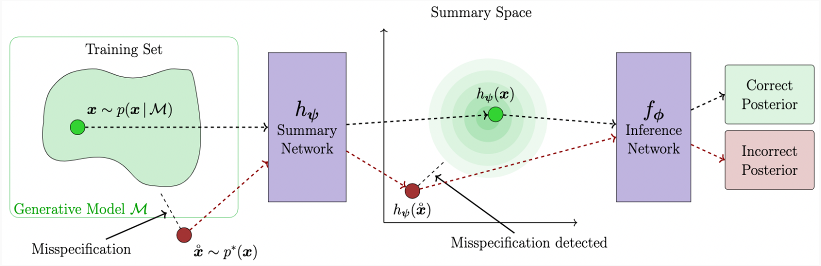
```
data = dict(y = ...)
posterior = approximator.sample(1000, conditions=data)

fig=bf.diagnostics.plots.pairs_posterior(
  estimates=posterior)
```

Diagnostics

Posterior predictive checks

- Simulate data from $p(\theta \mid y^{\text{obs}})p(y \mid \theta)$
- Compare simulated data to observed data
    - Visual checks - histograms, ecdf plots, scatter plots,...
    - Comparing statistics - means, variances, ...

Test for distribution shift



Training Set
$\boldsymbol{x} \sim p(\boldsymbol{x} \mid \mathcal{M})$
Generative Model $\mathcal{M}$
Misspecification
$\mathring{\boldsymbol{x}} \sim p^*(\boldsymbol{x})$
h_ψ Summary Network
Summary Space
$h_\psi(\boldsymbol{x})$
$h_\psi(\mathring{\boldsymbol{x}})$
Misspecification detected
f_ϕ Inference Network
Correct Posterior
Incorrect Posterior