Bayesian Semiparametric Regression with Liesel 04.12.2023
Tutorial at CMStatistics, Berlin

Exercise Sheet 01

Building on the concepts introduced in our previous session, this sheet will guide you through the practical application of Markov Chain Monte Carlo (MCMC) methods using the Goose framework to obtain samples from a posterior distribution. In this exercise, we will delve deeper into the analysis of parameters $\boldsymbol{\theta} = (\mu, \sigma)^T$. In contrast to the last exercise sheet, we will not consider the posterior mode but focus on the posterior mean $\mathrm{E}(\boldsymbol{\theta}|\boldsymbol{y})$ as well as considering the marginal 90% credible intervals. To facilitate using the Hamiltonian Monte Carlo (HMC) method, which is a gradient-based sampler, it is required that all parameters are continuous and encompass the entirety of real values. This requirement leads us to model σ in terms of $\log(\sigma)$, mirroring the approach taken in the last exercise. Before we proceed, let's revisit and align our understanding of how we could have defined the model in the second exercise of the previous sheet. This will ensure that everyone is on the same page.

In this context, we will employ a class derived from `NamedTuple` to represent our parameters. This approach enhances clarity of the code and reduces the chance of errors in the implementation of the log unnormalized posterior, as it eliminates the need to remember the specific meaning of each entry in a parameter vector.

Below is an exemplar code snippet from solving the previous exercise that you can use to generate the data and define the model.

```python
# First, we load the modules required and sample the data.
import jax
import jax.numpy as jnp
import tensorflow_probability.substrates.jax as tfp

key = jax.random.PRNGKey(0)
mu = -1.4
sigma = 0.8
n = 10

dist = tfp.distributions.Normal(mu, 0.8**2)
obs = dist.sample(n, seed=key)

# Now we define the model.
# Note that we use a custom parameter class derived from NamedTuple
# instead of a parameter vector. This makes implementing the
# log unnormalized posterior less error-prone as we do not need to
# remember which entry in the parameter vector means what.

from typing import NamedTuple, TypeAlias
Array: TypeAlias = jax.Array
```

```
class Params2(NamedTuple):
    mu: float
    log_sigma: float

def log_prior2(params: Params2) -> Array:
    sigma = jnp.exp(params.log_sigma)
    lp_mu = tfp.distributions.Normal(0, 10).log_prob(params.mu,)
    lp_sigma =  tfp.distributions.HalfCauchy(0, 1).log_prob(sigma) + params.log_sigma
    return lp_mu + lp_sigma

def log_likelihood2(params: Params2, obs: Array) -> Array:
    sigma = jnp.exp(params.log_sigma)
    dist = tfp.distributions.Normal(params.mu, sigma)
    lps = dist.log_prob(obs)
    return jnp.sum(lps)

def log_uposterior2(params: Params2, obs: Array) -> Array:
    return log_prior2(params) + log_likelihood2(params, obs)
```

Using the model as defined here, let's create an MCMC scheme to obtain samples from the posterior using the Goose library and following a series of structured steps. This process is similar to what we covered in the lecture, but here you will implement it hands-on.

Steps to Build the Sampler:

1. **Import the Goose Library:** Begin by importing the Goose library. This can be done using the command:

```
import liesel.goose as gs
```

2. **Create the EngineBuilder:** Instantiate an `EngineBuilder` object which will be used to configure the sampling process.
3. **Configure the Engine:** The engine builder helps us to configure the engine (which will organize the sampling). Using the engine builder is neat, as it allows us to configure the engine in multiple small steps instead of doing it all in one construtor with many many parameters.
    - **Connect the Engine with the Model:** Use the builder's `set_model` method to connect the engine to your model. The appropriate interface to with a

NamedTuple is the NamedTupleInterface[1].

- **Set the Initial State:** The sampler needs an initial state to start from. Use the `set_initial_values` method and supply an instance of `Params2` with initial values.
- **Determine the Sampling Duration:** Specify how many samples you require and the warmup scheme. This is most simply done using the `set_duration` method. This method creates a warmup scheme similar to that in Stan when using the HMC or NUTS kernel. In our solution for this exercise, we use `warmup_duration=500` and `posterior_duration=1000`s.
- **Configure Sampling Kernels:** Assign kernels to the parameters. For simplicity, use the `HMCKernel` for the entire parameter vector/object. Add a kernel by using the builder's method `add_kernel` and provide an instance of the desired kernel.

4. **Build the Engine:** Once the engine is fully configured, use the `build` method to create and return the engine. This engine can then be used for sampling.
5. **Run the MCMC Sampler:** Now that we have the engine configured and created, we can use the method `sample_all_epochs` to draw MCMC samples for all chains in parallel. You will note from the output that there are multiple warmup phases and one posterior phase. In the Liesel lingo, we call these phases epochs.
6. **Summarize the Posterior Samples:** Goose provides some tools to summarize the results and perform some diagnostics. The most straightforward to use is the `Summary` class. When printed, this class gives you a nice summary table but it also allows programatic access to all properties, which is in particular helpful when summarizing the results of a simulation study. The constructor of `Summary` expects an `SamplingResults` object which you can extract from the engine using the method get_results.

   In the result table you can find the posterior mean as well as the credible interval for each parameter. Moreover, the summary provides a table summarizing errors or warnings encountered during sampling.
7. **[Optional] Graphical Tools** Goose also has some graphical tools that you can use. For instance `plot_trace` might help you to assess convergence of the MCMC chain. Please feel free to explore the plotting functions (they start with `plot_`.) in the module `liesel.goose`.

[1]Alternatives you might want to use with your own model implementation are the `DictInterface` and `DataclassInterface` when using a dictionary or dataclass to represent the parameter vector, respectively. When connecting Goose to a Liesel model, the `LieselInterface` class should be used and for a PyMC model the connection is done with the `PyMCInterface`.

8. **[Optional] Summary of a Transformed Quantity** Even though, we defined the model in terms of $\log(\sigma)$, we might be more interested in the posterior distribution of σ. Since the transformation used is non-linear, we cannot just apply exp to the posterior mean or the credible interval borders to find the equivalent values for σ. Instead, we need to apply the transformation to each sample of $\log(\sigma)$ and subsequently calculate the summary statistics for the transformed quantity. One way yo do this in Goose is to provide the `additional_chain` argument to the constructor of `Summary`. The `additional_chain` is a dictionary where the keys are strings with the name of the transformed quantity. Its values are the posterior samples of the MCMC chain. Note that those need to have the same shape as the posterior samples of the originally sampled model parameters.

   To perform a transformation on model parameters, we can extract the chain from the `SamplingResults` object using the method `get_posterior_samples`. This returns a dictionary, where the keys refer to the model parameter names and the values are an `jax.Array` with the posterior sample. Thus, one could apply `jnp.exp` on the array associated with $\log \sigma$, and use these values to build the `additional_chain` argument.

   Note, when using Liesel's model library, this step become more simple (next sheet).

   We want to obtain the posterior distribution of σ from our model defined in terms of $\log(\sigma)$. Due to the non-linear transformation, directly applying exp to the posterior mean or credible interval limits of $\log(\sigma)$ is not appropriate for obtaining equivalent values for σ. Instead, each sample of $\log(\sigma)$ must be individually transformed, and summary statistics should then be calculated for this transformed data. In Goose, this can be achieved by using the `additional_chain` argument in the `Summary` constructor. This argument takes a dictionary where keys are the names of the transformed quantities, and their values are a `jax.Array` with the respective posterior samples from the MCMC chain. These samples must match the shape of the originally sampled model parameters.

   To transform model parameters, extract the posterior chain from the `SamplingResults` object using `get_posterior_samples`. This method returns a dictionary with keys corresponding to model parameter names and values being a `jax.Array` of the posterior samples. Applying `jnp.exp` to the array for $\log \sigma$, gives us the posterior samples σ which can be used as the values in a dictionary supplied as `additional_chain`.

   Note that using Liesel's model library simplifies this process, as will be discussed in the next sheet.