Measurement Error Models

General Principles

Measurement error refers to the variability in the measurement of a variable, and measurement error can be generated by several factors, such as sampling bias, censoring bias, and group size heterogeneity. It is an important consideration in many fields, including statistics, economics, and engineering, where accurate measurements are crucial for making informed decisions. To account for measurement error, we can use a *measurement error model*. This model assumes that the measurement of a variable is subject to an error, which can be modeled using a probability distribution. The model can be used to estimate the parameters of the measurement error distribution, such as the mean and variance, and to make predictions about the measurements based on the estimated parameters. Measurement error models are *composed models* (i.e., models with sub-models) that evaluate different generative processes, starting with the measurement error process, which is then used to generate the observed data.

Example

Below is an example code snippet demonstrating a Bayesian measurement error model using the Bayesian Inference (BI) package. The data consist of three continuous variables (marriage rate, divorce rate, age), and the goal is to estimate the effect of age and marriage rate on the divorce rate while considering that the divorce rate has a measurement error. This example is based on McElreath (2018).

Python

```
from BI import bi, jnp

# Setup device--------------------------------------------------
m = bi(platform='cpu')

# Import Data & Data Manipulation ------------------------------------------------
```

```python
# Import
data_path = m.load.WaffleDivorce(only_path=True)
m.data(data_path, sep=';')
m.scale(['MedianAgeMarriage', 'Marriage']) # Scale
dat = dict(
    D_obs = m.z_score(m.df['Divorce'].values),
    D_sd = jnp.array(m.df['Divorce SE'].values / m.df['Divorce'].std()),
    A = jnp.array(m.df['MedianAgeMarriage'].values),
    M = jnp.array(m.df['Marriage'].values),
    N = m.df.shape[0]
)
m.data_on_model = dat # Send to model (convert to jax array)

# Define model ------------------------------------------------
def model(D_obs, D_sd, A, N, M):
    a = m.dist.normal(0, 0.2, name = 'a')
    beta = m.dist.normal(0, 0.5, name = 'beta')
    eta = m.dist.normal(0, 0.5, name = 'eta')
    s = m.dist.exponential(1, name = 's')
    mu = a + beta * A + eta * M
    D_true = m.dist.normal(mu, s, name = 'D_true')
    m.dist.normal(D_true , D_sd, obs = D_obs)

# Run MCMC ----------------------------------------------------
m.fit(model)  # Optimize model parameters through MCMC sampling

# Summary -----------------------------------------------------
m.summary() # Get posterior distributions
```

```
jax.local_device_count 32
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]warmup:   0%|          | 1/1000 [00:00<12:23,  1.34it
arviz - WARNING - Shape validation failed: input_shape: (1, 500), minimum_shape: (chains=2, d
```

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
D_true[0]	1.20	0.35	0.73	1.83	0.01	0.02	594.50	311.15	NaN
D_true[1]	0.72	0.56	-0.18	1.54	0.02	0.03	825.18	357.15	NaN
D_true[2]	0.46	0.32	-0.00	0.99	0.01	0.02	804.31	242.18	NaN
D_true[3]	1.44	0.47	0.65	2.05	0.02	0.03	761.95	307.56	NaN
D_true[4]	-0.91	0.13	-1.13	-0.70	0.00	0.01	988.33	321.69	NaN

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
D_true[5]	0.66	0.41	0.04	1.36	0.01	0.02	1246.87	323.91	NaN
D_true[6]	-1.39	0.38	-1.89	-0.76	0.01	0.02	1009.18	298.94	NaN
D_true[7]	-0.35	0.52	-1.17	0.49	0.02	0.03	1089.02	423.66	NaN
D_true[8]	-1.91	0.62	-2.89	-0.98	0.02	0.03	679.08	393.51	NaN
D_true[9]	-0.63	0.17	-0.87	-0.36	0.00	0.01	1349.49	348.55	NaN
D_true[10]	0.78	0.27	0.38	1.22	0.01	0.01	1051.89	257.32	NaN
D_true[11]	-0.55	0.50	-1.40	0.22	0.02	0.02	847.21	393.66	NaN
D_true[12]	0.14	0.55	-0.69	0.99	0.02	0.03	695.14	205.29	NaN
D_true[13]	-0.89	0.24	-1.25	-0.51	0.01	0.01	1197.99	353.12	NaN
D_true[14]	0.57	0.28	0.11	1.00	0.01	0.01	1263.50	425.36	NaN
D_true[15]	0.29	0.34	-0.20	0.84	0.01	0.02	902.36	369.49	NaN
D_true[16]	0.50	0.42	-0.09	1.20	0.01	0.02	918.81	387.37	NaN
D_true[17]	1.29	0.37	0.78	1.93	0.01	0.02	1072.81	329.82	NaN
D_true[18]	0.46	0.39	-0.13	1.09	0.01	0.02	858.17	321.87	NaN
D_true[19]	0.44	0.56	-0.49	1.26	0.03	0.02	483.84	347.58	NaN
D_true[20]	-0.57	0.29	-1.06	-0.13	0.01	0.01	840.30	406.25	NaN
D_true[21]	-1.11	0.26	-1.56	-0.75	0.01	0.02	1349.49	173.98	NaN
D_true[22]	-0.27	0.25	-0.63	0.13	0.01	0.01	1306.06	365.28	NaN
D_true[23]	-1.02	0.29	-1.48	-0.57	0.01	0.02	1007.31	270.10	NaN
D_true[24]	0.46	0.40	-0.28	0.98	0.01	0.02	1027.80	353.03	NaN
D_true[25]	-0.03	0.28	-0.52	0.37	0.01	0.01	1017.75	393.51	NaN
D_true[26]	-0.04	0.53	-0.88	0.76	0.02	0.03	1122.42	305.21	NaN
D_true[27]	-0.16	0.39	-0.71	0.48	0.01	0.02	1349.49	351.46	NaN
D_true[28]	-0.25	0.52	-1.03	0.61	0.02	0.05	966.76	160.75	NaN
D_true[29]	-1.83	0.26	-2.20	-1.40	0.01	0.01	1106.20	352.03	NaN
D_true[30]	0.19	0.48	-0.60	0.93	0.01	0.03	1349.49	302.06	NaN
D_true[31]	-1.67	0.16	-1.96	-1.43	0.00	0.01	1115.72	284.32	NaN
D_true[32]	0.12	0.23	-0.25	0.48	0.01	0.01	1187.57	311.15	NaN
D_true[33]	-0.08	0.49	-0.79	0.76	0.02	0.03	833.38	302.17	NaN
D_true[34]	-0.11	0.25	-0.48	0.30	0.01	0.02	1349.49	219.22	NaN
D_true[35]	1.31	0.44	0.62	2.04	0.01	0.02	1131.53	353.12	NaN
D_true[36]	0.24	0.34	-0.32	0.74	0.01	0.02	790.58	383.87	NaN
D_true[37]	-1.04	0.21	-1.38	-0.73	0.01	0.01	1349.49	365.09	NaN
D_true[38]	-0.91	0.55	-1.92	-0.16	0.02	0.03	767.57	425.43	NaN
D_true[39]	-0.71	0.29	-1.12	-0.24	0.01	0.01	945.80	438.47	NaN
D_true[40]	0.26	0.53	-0.64	1.09	0.02	0.03	1201.76	255.08	NaN
D_true[41]	0.76	0.33	0.14	1.17	0.01	0.02	763.53	224.82	NaN
D_true[42]	0.20	0.17	-0.11	0.46	0.00	0.01	1349.49	461.64	NaN
D_true[43]	0.80	0.48	0.07	1.56	0.02	0.02	931.02	304.36	NaN
D_true[44]	-0.40	0.52	-1.27	0.35	0.01	0.03	1349.49	368.12	NaN
D_true[45]	-0.39	0.24	-0.74	-0.02	0.01	0.01	933.88	332.15	NaN

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
D_true[46]	0.13	0.31	-0.30	0.67	0.01	0.02	1349.49	294.58	NaN
D_true[47]	0.60	0.47	-0.15	1.36	0.01	0.03	1337.89	292.95	NaN
D_true[48]	-0.64	0.29	-1.07	-0.13	0.01	0.02	1349.49	337.37	NaN
D_true[49]	0.88	0.62	-0.11	1.85	0.02	0.03	817.15	370.41	NaN
a	-0.05	0.10	-0.21	0.11	0.00	0.00	700.95	468.17	NaN
beta	-0.62	0.16	-0.88	-0.37	0.01	0.01	407.98	254.65	NaN
eta	0.05	0.17	-0.18	0.36	0.01	0.01	370.43	429.80	NaN
s	0.60	0.11	0.44	0.78	0.01	0.00	325.49	352.19	NaN

R

```
library(BayesianInference)
jnp = reticulate::import('jax.numpy')

# Setup platform-----------------------------------------------
m=importBI(platform='cpu')

# Import data -------------------------------------------------
m$data(paste(system.file(package = "BayesianInference"),"/data/WaffleDivorce.csv", sep = ''))

m$scale(list('MedianAgeMarriage', 'Marriage'))

m$data_on_model$D_obs = m$z_score(jnp$array(m$df['Divorce']))
m$data_on_model$D_sd = jnp$array(m$df['Divorce SE']) / sd(unlist(m$df['Divorce']))
m$data_on_model$A = jnp$array(m$df['MedianAgeMarriage'])
m$data_on_model$M = jnp$array(m$df['Marriage'])
m$data_on_model$N = as.integer(nrow(m$df))

# Define model ------------------------------------------------
model <- function(D_obs, D_sd, A, N, M){
  a = bi.dist.normal(0, 0.2, name = 'a')
  beta = bi.dist.normal(0, 0.5, name = 'beta')
  eta = bi.dist.normal(0, 0.5, name = 'eta')
  s = bi.dist.exponential(1, name = 's')
  mu = a + beta * A + eta * M
  D_true = bi.dist.normal(mu, s, name = 'D_true')
  bi.dist.normal(D_true , D_sd, obs = D_obs)
}
```

```
# Run MCMC -----------------------------------------------
m$fit(model) # Optimize model parameters through MCMC sampling

# Summary ------------------------------------------------
m$summary() # Get posterior distribution
```

Julia

```
using BayesianInference

# Setup device-------------------------------------------------
m = importBI(platform="cpu")

# Import Data & Data Manipulation ------------------------------------------------
# Import
data_path = m.load.WaffleDivorce(only_path=true)
m.data(data_path, sep=";")
m.scale(["MedianAgeMarriage", "Marriage"]) # Scale
dat = pydict(
    D_obs = m.z_score(m.df["Divorce"].values),
    D_sd = jnp.array(m.df["Divorce SE"].values / m.df["Divorce"].std()),
    A = jnp.array(m.df["MedianAgeMarriage"].values),
    M = jnp.array(m.df["Marriage"].values),
    N = m.df.shape[0]
)
m.data_on_model = dat # Send to model (convert to jax array)

# Define model -------------------------------------------------
@BI function model(D_obs, D_sd, A, N, M)
    a = m.dist.normal(0, 0.2, name = "a")
    beta = m.dist.normal(0, 0.5, name = "beta")
    eta = m.dist.normal(0, 0.5, name = "eta")
    s = m.dist.exponential(1, name = "s")
    mu = a + beta * A + eta * M
    D_true = m.dist.normal(mu, s, name = "D_true")
    m.dist.normal(D_true , D_sd, obs = D_obs)


end

# Run mcmc -----------------------------------------------
```

```
m.fit(model)  # Optimize model parameters through MCMC sampling

# Summary ----------------------------------------------
m.summary() # Get posterior distributions
```

Mathematical Details

Bayesian formulation

$$D_i^* \sim \text{Normal}(D_i, \varsigma_i)$$

$$D_i \sim \text{Normal}(\mu_i, \sigma)$$

$$\mu_i = \alpha + \beta A_i + \eta M_i$$

$$\sigma \sim \text{Normal}(1)$$

where:

- D_i^* is the observed divorce rate.
- D_i is the true divorce rate.
- μ_i is the mean of the true divorce rate.
- σ is the standard deviation of the true divorce rate.
- α is the intercept term.
- β is the regression coefficient for age.
- η is the regression coefficient for marriage rate.

Notes

> **Note**
>
> This is an approach that can be extended to any kind of model previously described. For example, one could generate a Bernoulli measurement error model by generating a process for the probabilities of success and failure. We can even go further by potentially having an error rate that is present only in one of the two outcomes.

Reference(s)

McElreath, Richard. 2018. *Statistical Rethinking: A Bayesian course with examples in R and Stan*. Chapman; Hall/CRC.