BNPlib: A Nonparametric C++ Library

Bruno Guindani
Elena Zazzetti

July 9th, 2020

POLITECNICO
MILANO 1863

https://github.com/poliprojects/BNPlib

Introduction

Bayesian approach

- parameter ϑ: is a *random variable*
- dataset $\boldsymbol{y}$
- **Bayes Theorem**:

$$\left.\begin{array}{c} \pi(\vartheta) \text{ prior distribution} \\ p(\boldsymbol{y}|\vartheta) \text{ likelihood} \end{array}\right\} \Rightarrow \pi(\vartheta|\boldsymbol{y}) \text{ posterior distribution}$$

$$\pi(\vartheta|\boldsymbol{y}) = \frac{p(\boldsymbol{y}|\vartheta)\ \pi(\vartheta)}{\int_\Theta p(\boldsymbol{y}|\vartheta)\ \pi(\vartheta)\,\mathrm{d}\vartheta}$$

Markov Chain Monte Carlo (MCMC) methods

Approximate the integral

$$\mathcal{I} = \int h(x) f(x) \mathrm{d}x$$

$y_1, \ldots, y_n \sim f$ without directly simulating from $f \Rightarrow$ Markov chain with stationary distribution f

- One algorithm for constructing chains: Gibbs Sampling.

Non-parametric Bayesian model

Having observed the iid sample $\{y_i\}_i$, $i = 1, \ldots, n$:

Parametric model:

$$y_1, \ldots, y_n \mid \vartheta \overset{\text{iid}}{\sim} f(y \mid \vartheta)$$
$$\vartheta \sim \pi$$

Non-parametric model:

$$y_1, \ldots, y_n \mid G \overset{\text{iid}}{\sim} G$$
$$G \sim \pi$$

DP and DPM models

- Dirichlet process model (discrete):

$$y_i | G \overset{\text{iid}}{\sim} G$$
$$G \sim DP(MG_0)$$

- Dirichlet process mixture (**DPM**) model (continuous):

$$y_i | G \overset{\text{iid}}{\sim} f_G(\cdot) = \int_\Theta f(\cdot|\boldsymbol{\vartheta})\, G(\mathrm{d}\boldsymbol{\vartheta})$$
$$G \sim DP(MG_0)$$

$$\iff$$

$$y_i | \boldsymbol{\vartheta}_i \overset{\text{ind}}{\sim} f(\cdot|\boldsymbol{\vartheta}_i)$$
$$\boldsymbol{\vartheta}_i | G \overset{\text{iid}}{\sim} G$$
$$G \sim DP(MG_0)$$

- $\boldsymbol{\vartheta}_i$ latent variables
- G is discrete $\implies$ ties: **allocations** c_i, **unique values** ϕ_{c_i}
- These form the *state* of the algorithm

Case studies: hierarchies

- Normal Normal-InverseGamma (**NNIG**) hierarchy:

$$f(y|\boldsymbol{\vartheta}) = N(y|\mu, \sigma^2)$$

$$G_0(\boldsymbol{\vartheta}|\mu_0, \lambda_0, \alpha_0, \beta_0) = N\left(\mu|\mu_0, \frac{\sigma^2}{\lambda_0}\right) \times \text{Inv-Gamma}(\sigma^2|\alpha_0, \beta_0)$$

  with latent variables $\boldsymbol{\vartheta} = (\mu, \sigma)$

- Normal Normal-Wishart (**NNW**) hierarchy:

$$f(\mathbf{y}|\boldsymbol{\vartheta}) = N(\mathbf{y}|\boldsymbol{\mu}, T^{-1}),$$

$$G_0(\boldsymbol{\vartheta}|\boldsymbol{\mu}_0, \lambda, T_0, \nu) = N\left(\mu|\boldsymbol{\mu}_0, (\lambda T)^{-1}\right) \times \text{Wish}(T|T_0, \nu).$$

  with latent variables $\boldsymbol{\vartheta} = (\boldsymbol{\mu}, T)$

Algorithms

Neal's Algorithm 2

- Gibbs sampler for conjugate models
- $(\boldsymbol{\phi}, \mathbf{c})$ is the **state** of a Markov chain
- **Allocation sampling**:
  For $i = 1, \ldots, n$: update c_i
  - If c_i allocates ϕ_{c_i} to a singleton, remove ϕ_{c_i} from the state
  - Sample c_i as follows:

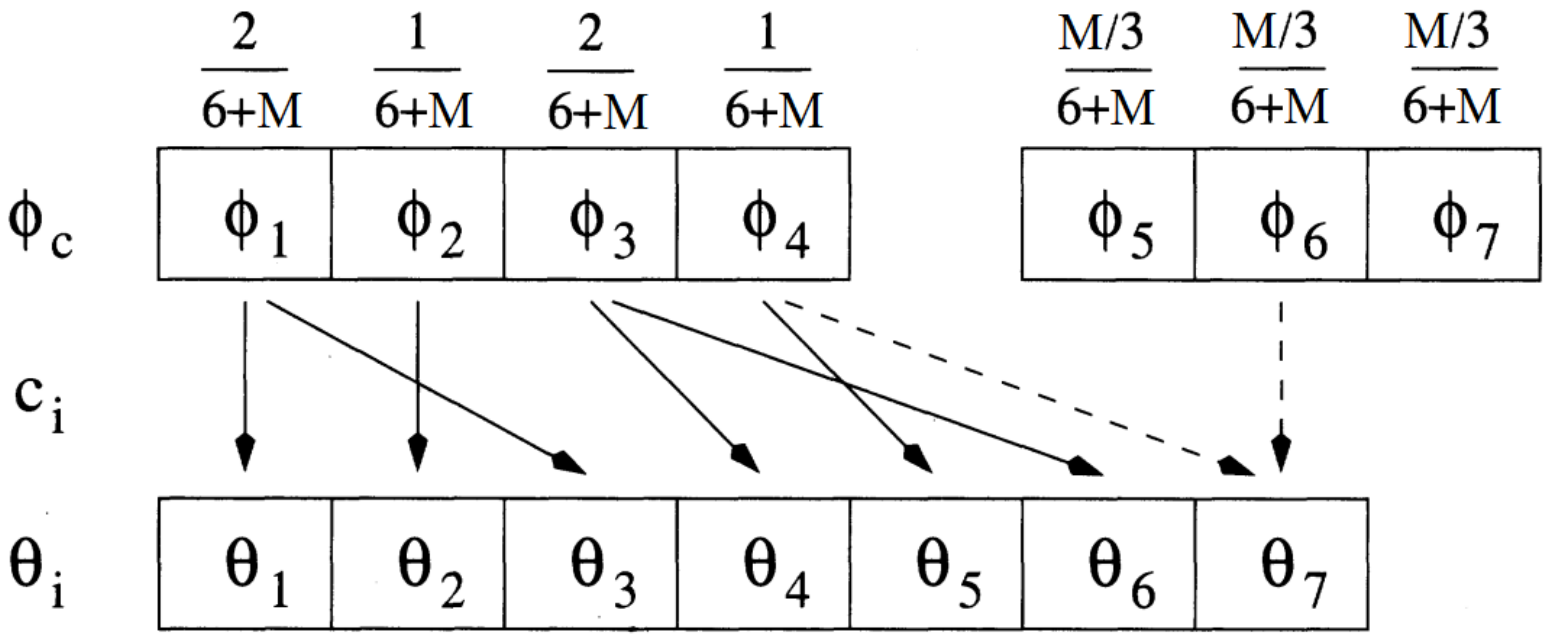
$$\text{If } c = c_j \text{ for some } j \neq i\text{: } \mathbb{P}(c_i = c | \mathbf{c}_{-i}, y_i, \boldsymbol{\phi}) \propto \frac{n_{-i,c}}{n - 1 + M} F(y_i, \phi_c)$$

$$\text{total } \mathbb{P}(c_i \neq c_j \text{ for all } j | \mathbf{c}_{-i}, y_i, \boldsymbol{\phi}) \propto \frac{M}{n - 1 + M} \int F(y_i, \phi)\, G_0(\mathrm{d}\phi)$$

  - If the new c_i allocates ϕ_{c_i} to a singleton, draw $\phi_{c_i} \sim G_0(\cdot | y_i)$ and add it to the state
- **Unique values sampling**:
  For $c \in \{c_1, \ldots, c_n\}$: update ϕ_c, given all the y_i with $c_i = c$

Neal's Algorithm 8

Gibbs sampling on the state, which is extended by the addition of m **auxiliary blocks** (here $m = 3$, on the right)



Neal's Algorithm 8

- **Allocation sampling**:
  For $i = 1, \dots, n$: update c_i
  - Sample auxiliary parameters:
    - $c_i = c_j$ for some $j \Rightarrow$ no connection
    - $c_i \neq c_j \Rightarrow$ association to one of m

    The other ϕ values drawn from G_0
  - Draw c_i as follows:

$$P(c_i = c | \mathbf{c}_{-i}, y_i, \phi_1, \dots, \phi_h) \propto \begin{cases} \frac{n_{-i,c}}{n-1+M} F(y_i, \phi_c), & \text{for } 1 \leq c \leq k^- \\ \frac{M/m}{n-1+M} F(y_i, \phi_c), & \text{for } k^- + 1 < c \leq h \end{cases}$$

    with k^- unique values excluding c_i and $h = k^- + m$
  - Discard values in ϕ not associated to any ϑ_j
- **Unique values sampling**:
  For $c \in \{c_1, \dots, c_n\}$: update ϕ_c given y_i such that $c_i = c$

Implementation

Libraries

`bnplib` depends on:

- *Eigen*
- *Stan Math*
  - *Eigen*
  - *Boost*
  - Intel's *Threading Building Blocks* (TBB)
  - *Sundials*
- Google's *Protocol Buffer* (Protobuf)

Python interface uses:

- Protobuf
- *pybind11*
- *NumPy*
- *SciPy*
- *Scikit-learn*
- *Matplotlib*

Model classes

- `Mixture`:

BaseMixture
`mass_existing_cluster()`
`mass_new_cluster()`

  Derived: `DirichletMixture`, `PitYorMixture`

- `Hypers`: `HypersFixedNNIG`, `HypersFixedNNW`
- `Hierarchy`:

HierarchyBase<Hypers>
`state`
`*hypers`
`eval_marg()`
`like()`
`sample_given_data()`
`draw()`

  Derived: `HierarchyNNIG`, `HierarchyNNW`

Algorithm classes

```
template<template <class> class Hierarchy,
    class Hypers, class Mixture> class Algorithm
```

- Constructor:

```
Algorithm(const Hypers &hypers_, const Mixture &mixture_,
    const Eigen::MatrixXd &data_, const unsigned int init = 0);
```

- Method parameters:

```
unsigned int maxiter = 1000;
unsigned int burnin = 100;
```

- Data and value containers:

```
Eigen::MatrixXd data;
std::vector<unsigned int> cardinalities;
std::vector<unsigned int> allocations;
std::vector< Hierarchy<Hypers> > unique_values;
std::pair< Eigen::MatrixXd, Eigen::VectorXd > density;
Mixture mixture;
State best_clust;
```

Algorithm classes

```
void step(){
    sample_allocations();
    sample_unique_values();
    sample_weights();
    update_hypers();
}

void run(BaseCollector* collector){
    initialize();
    unsigned int iter = 0;
    collector->start();
    while(iter < maxiter){
        step();
        if(iter >= burnin){
            save_state(collector, iter);
        }
    iter++;
    }
     collector->finish();
}
```

Neal2

- Derived from `Algorithm` class
- Overriden methods:
  - `initialize()`
  - `sample_allocations()`
  - `sample_unique_values()`

Neal8

- Derived from `Neal2` class
- Additional members:

```
unsigned int n_aux = 3;
std::vector<Hierarchy<Hypers>> aux_unique_values;
```

- Overriden method:
  - `sample_allocations()`

Cluster Estimate

- Best clustering, i.e. best dissimilarity matrix $D^{(k)}$:

$$\hat{k} = \arg\min_k \left\| D^{(k)} - \bar{D} \right\|_F^2 = \arg\min_k \sum_{i,j} \left(D_{ij}^{(k)} - \bar{D}_{ij} \right)^2$$

- `Algorithm` method:

```
unsigned int cluster_estimate(BaseCollector* coll);
```

  - `Eigen::SparseMatrix` class
  - Save a `State` object into `best_clust` member

- Writing utility:

```
write_clustering_to_file(const std::string &filename);
```

Density Estimate

- Density:

$$\hat{f}(x) = \frac{1}{K} \sum_k \hat{f}^{(k)}(x) \quad \text{with}$$

$$\hat{f}^{(k)}(x) = \sum_j \frac{n_j^{(k)}}{M+n} f\left(x|\phi_j^{(k)}\right) + \frac{M}{M+n} \hat{m}(x)$$

- `Algorithm` method:

```
void eval_density(const Eigen::MatrixXd &grid,
    BaseCollector* coll);
```

  - `density_marginal_component()` subroutine, specific for each derived class
  - Save result into `density` member

- Writing utility:

```
write_density_to_file(const std::string &filename);
```

Collectors

Use Protobuf messages to store data:

```
message Par_Col {
    repeated double elems = 1;
}
message Param {
    repeated Par_Col par_cols = 1;
}
message UniqueValues {
    repeated Param params = 1;
}
message State {
    repeated int32 allocations = 1;
    repeated UniqueValues uniquevalues = 2;
}
```

Collectors

Collector types:

- `FileCollector`: saves `State` objects into binary file
- `MemoryCollector`: deque of `State` objects

Writing mode:

- `start()`
- `collect()`
- `close()`

Reading mode:

- `get_chain()`
- `get_next_state()`
- `get_state(unsigned int i)`

Algorithm factory

```
template<class AbstractProduct, typename... Args>
    class Factory
```

- For runtime choice of the algorithm
- Builder:

```
Builder = std::function<std::unique_ptr<
    AbstractProduct>(Args...)>
```

- Storage for algorithm builders:

```
std::map<Identifier, Builder> storage;
```

- Builders in the storage: `neal2`, `neal8`
- Variadic template for possibly different number of parameters as input in constructors

Python interface

```
PYBIND11_MODULE(bnplibpy, m){
    m.doc() = "Nonparametric library for cluster and
        density estimation";
    m.def("run_NNIG_Dir", &run_NNIG_Dir);
    ...
}
```

- Using *Numpy* for array structures
- `get_multidim_grid()`
- `deserialize()` (via *Protobuf*)
- `clust_rand_score()` (via *Scikit-learn*)

Plotting tools via *Matplotlib*:

- `chain_barplot()`
- `plot_density_points()`
- `plot_density_contour()`
- `plot_clust_cards()`.

Performance and optimization

Sparse vs dense matrices

- For memory usage reasons in `cluster_estimate()`
- If k equal clusters, the nonzero entries for a dissimilarity matrix are

$$k\left(\binom{n/k}{2} - n\right) = \frac{n^2}{2k} - \frac{n}{2} - nk \ll n^2$$

- Similar efficiency:

	sparse	dense
test 1	4.51696840e+07	4.15315598e+07
test 2	1.18633301e+09	9.17060859e+08
test 3	2.91540577e+07	3.00693867e+07
test 4	1.27339513e+08	1.22764857e+08
test 5	1.50049441e+08	1.37240555e+08
test 6	1.28818714e+08	1.16328396e+08

Alternative implementation

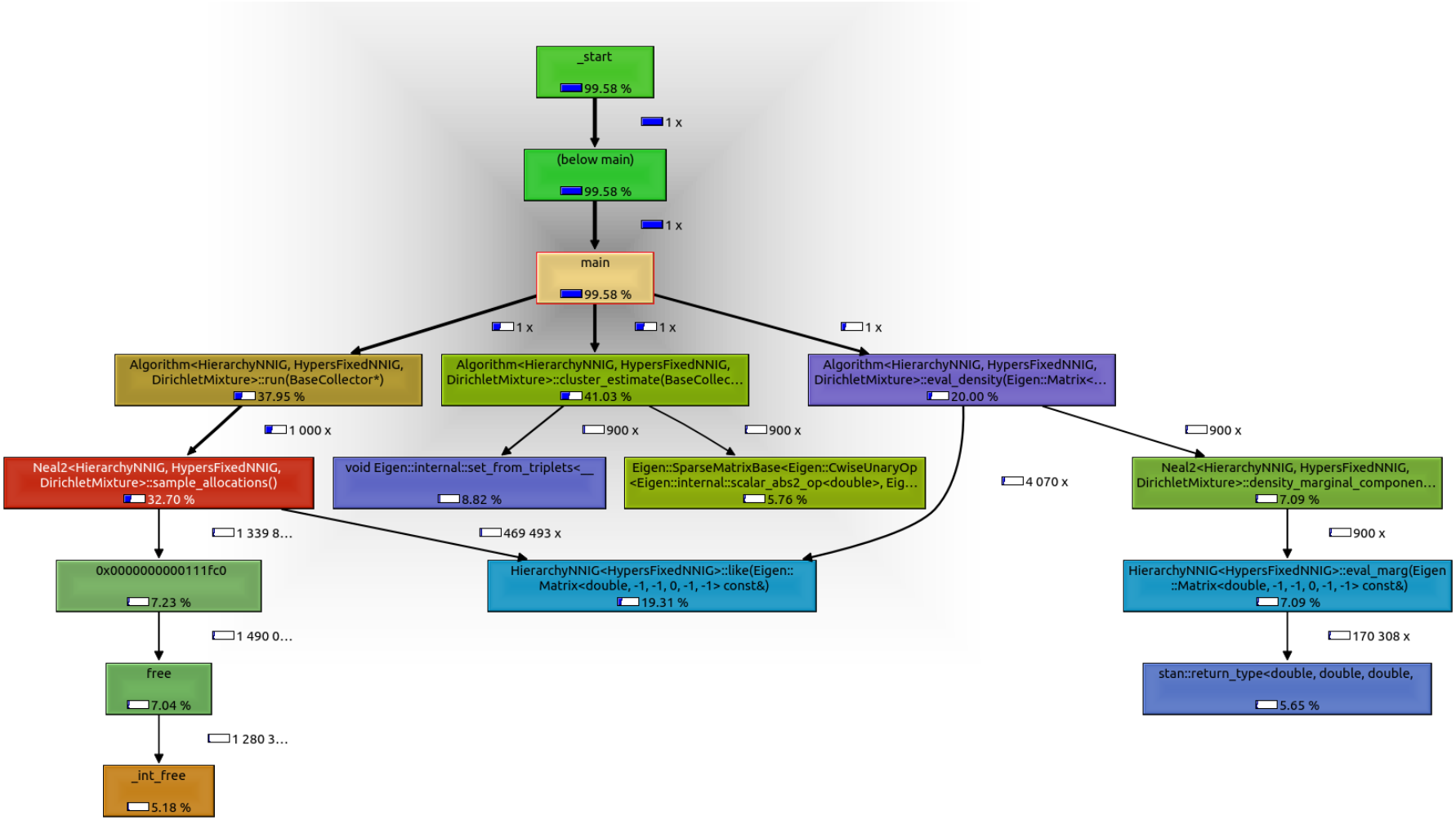
- Different approach of storing data, e.g. with an `std::map` of clusters
- A way to erase and add clusters more efficiently

Custom multivariate likelihood

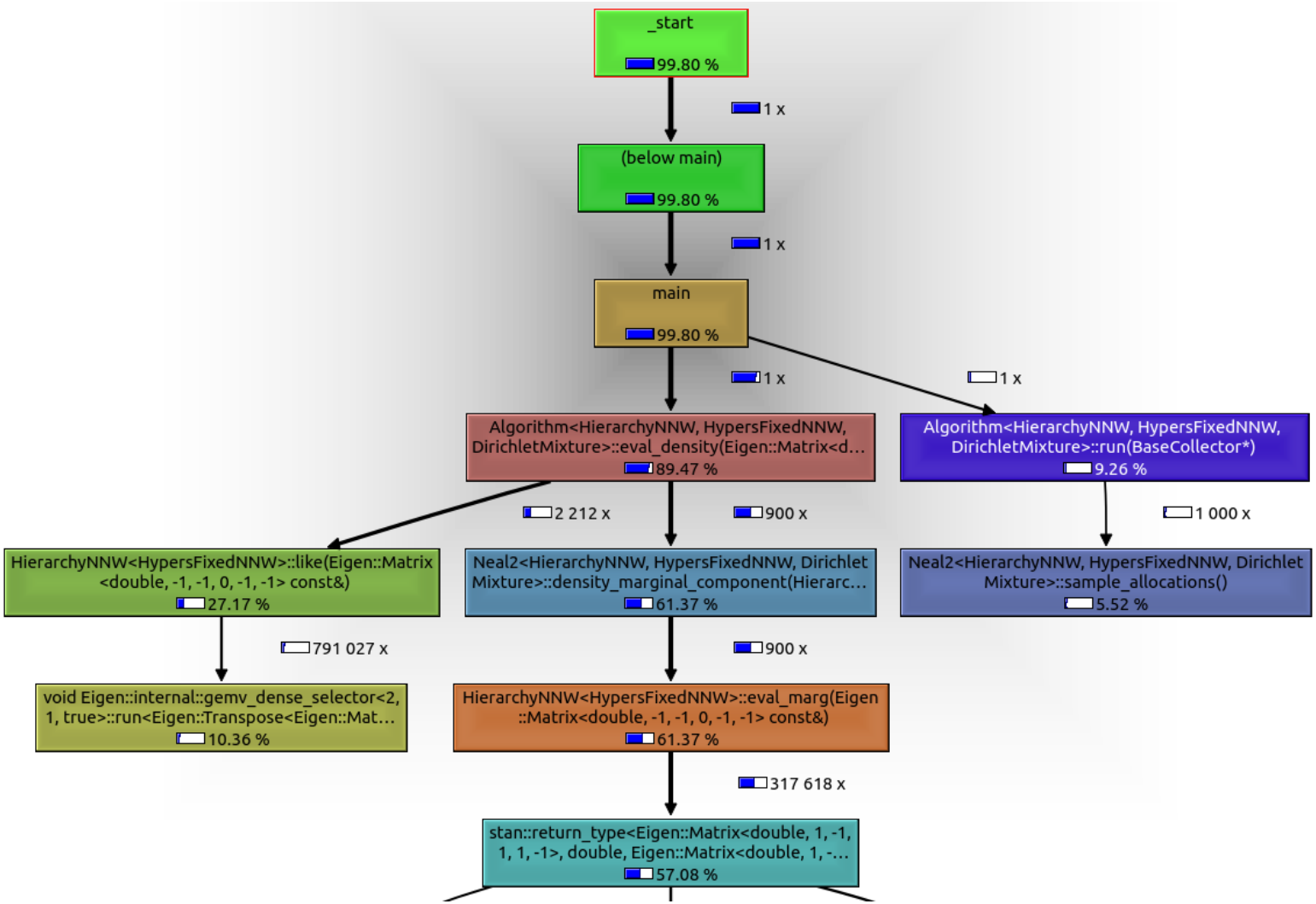
- In the `HierarchyNNW` class, as opposed to using the default Stan version
- Save matrix factorizations and determinants into the class to use them multiple times
- Improvement:

	`run()`	`eval_density()`	total
Stan	47700683	14317337	62018020
custom	39615848	10353269	49969117
speedup	1.20x	1.38x	1.25x

callgrind profiling: univariate



callgrind profiling: multivariate



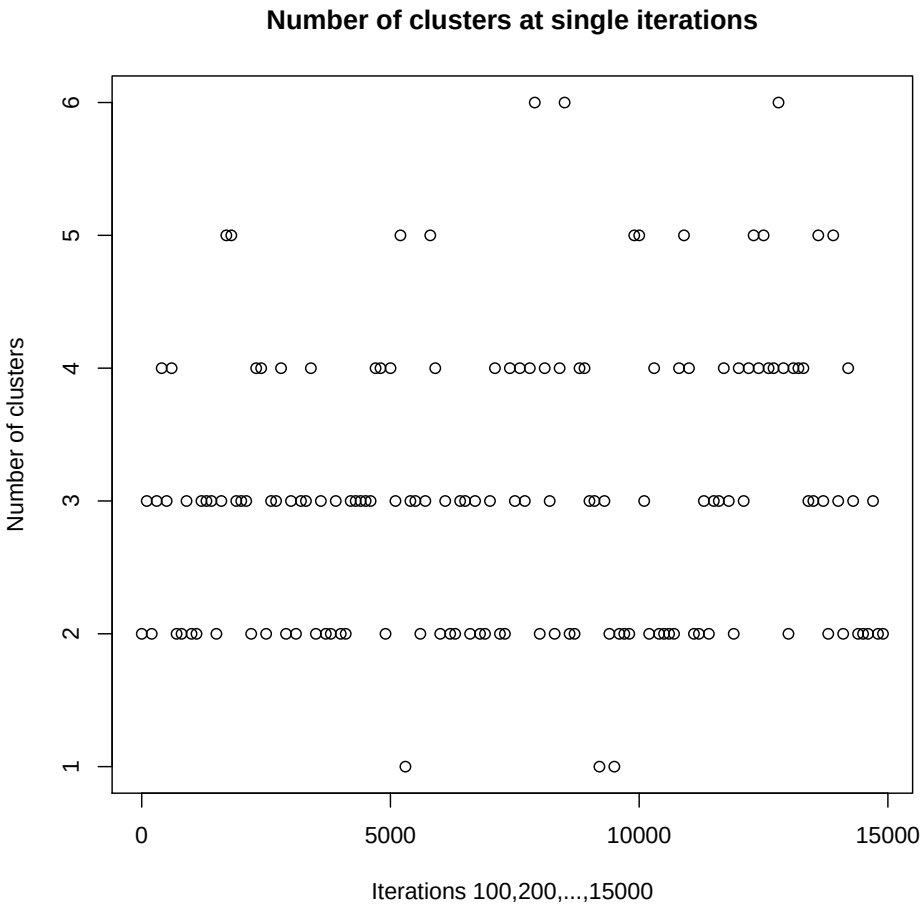
Results

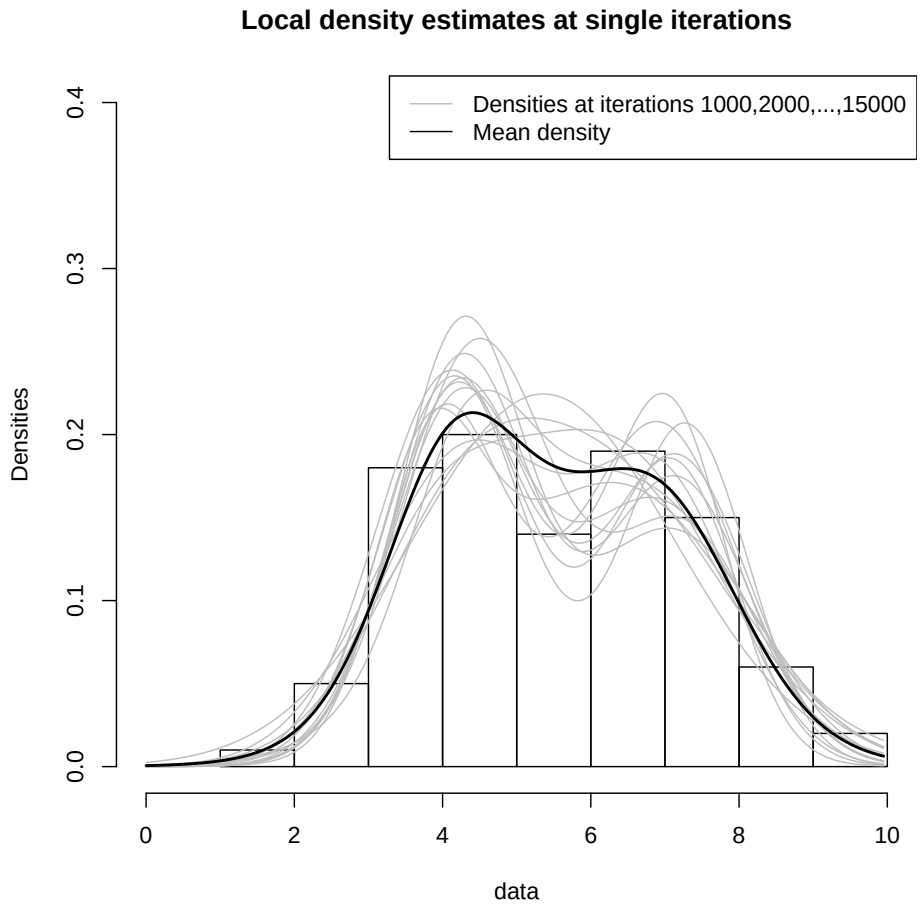
Sensitivity analysis

Set-up:

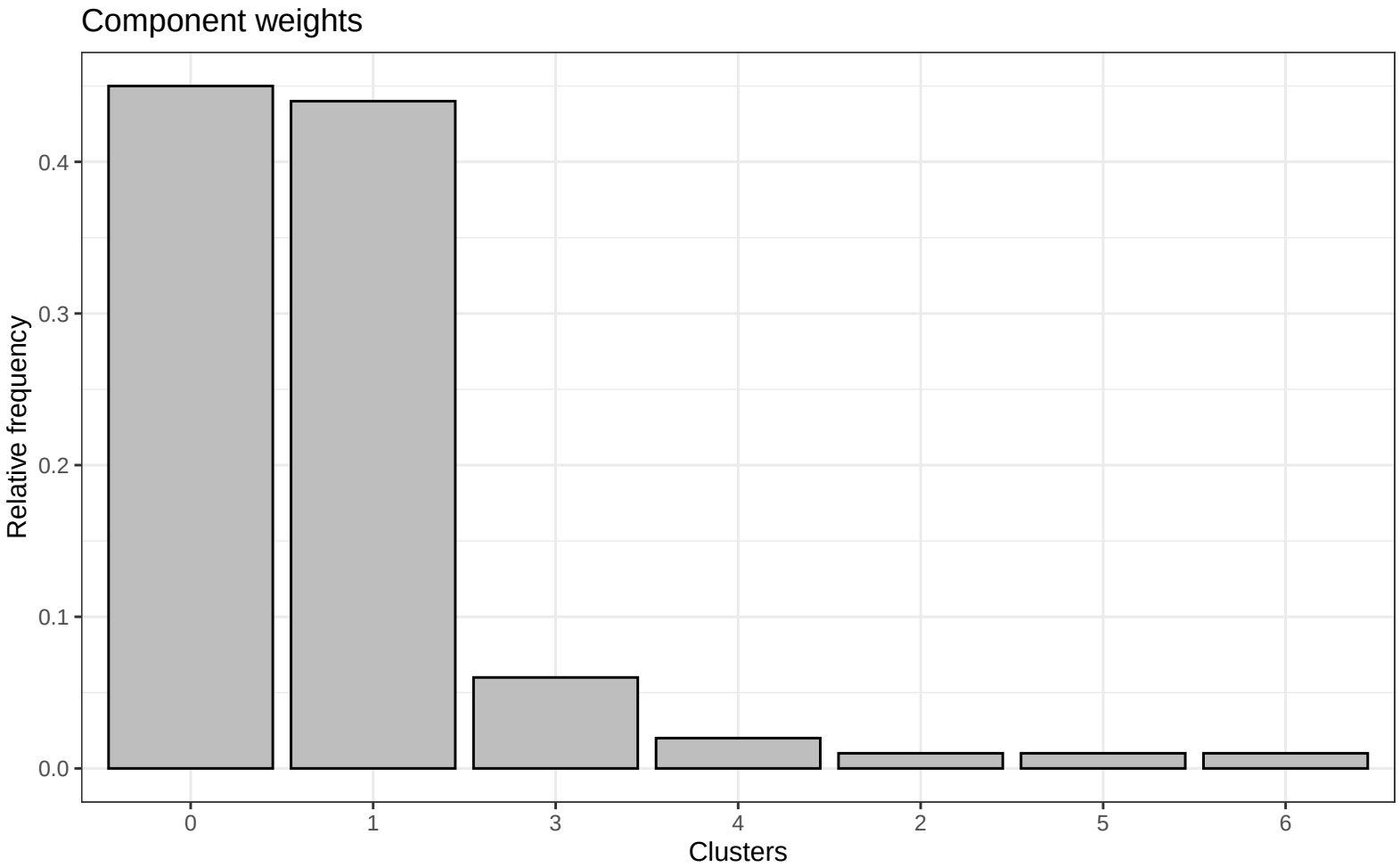
- $n = 100$ observations:
  - $y_1, \dots, y_{50} \overset{\text{iid}}{\sim} \mathcal{N}(4, 1)$
  - $y_{51}, \dots, y_{100} \overset{\text{iid}}{\sim} \mathcal{N}(7, 1)$
- Prior parameters for the Normal-NIG model:
  - $\mu_0 = 5$
  - $\lambda_0 = 1$
  - $\alpha_0 = 2$
  - $\beta_0 = 2$
- Dirichlet Mixture with variable M
- Method parameters:
  - $m = 3$ (for `Neal8`)
  - `maxiter` $= 20000$
  - `burnin` $= 5000$

Oscillations





Clustering


Component weights
Relative frequency
0.0
0.1
0.2
0.3
0.4
0
1
3
4
2
5
6
Clusters

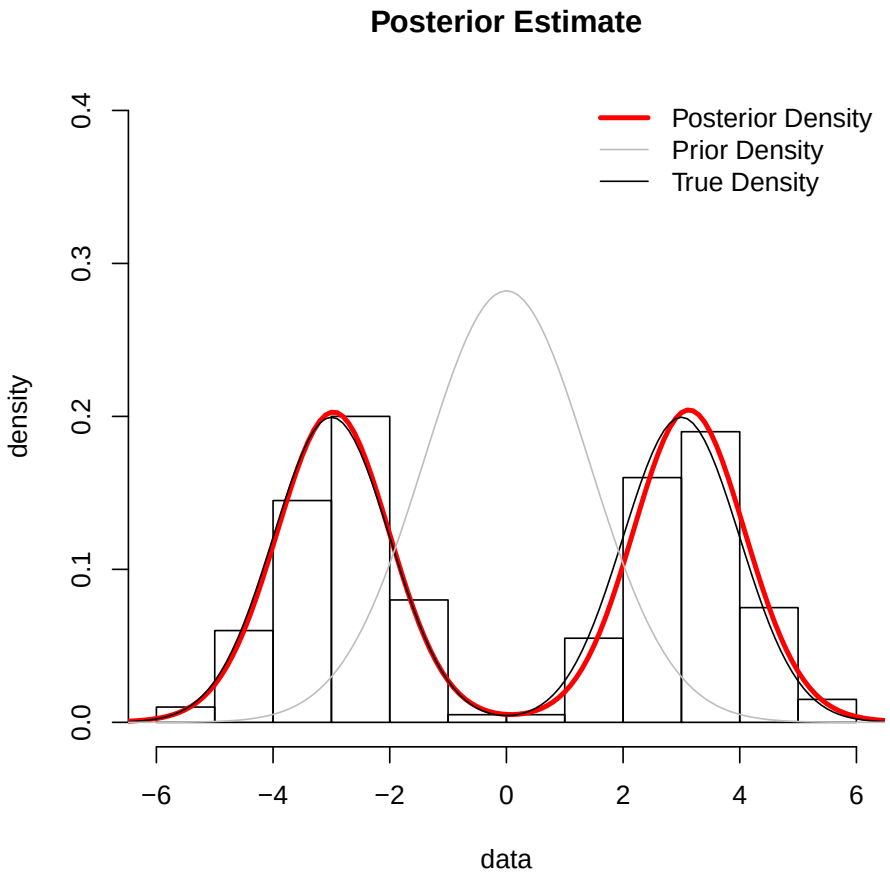
Python tests

Data:

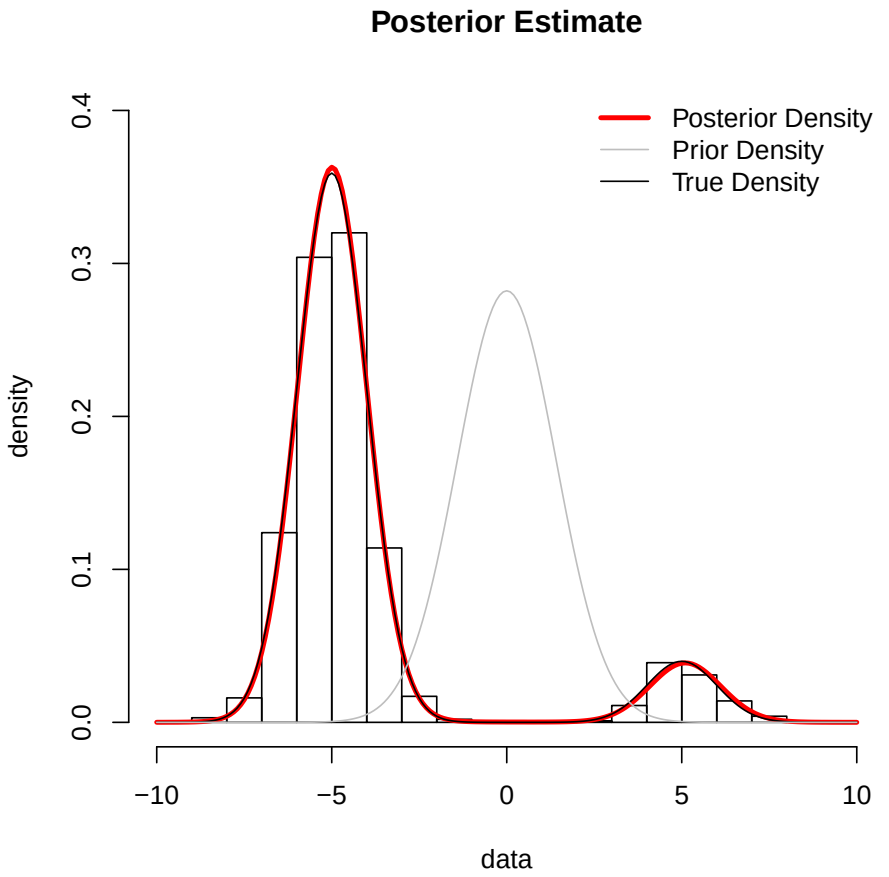
test	n	process
1	200	$y \sim 0.5\,\mathcal{N}(-3,1) + 0.5\,\mathcal{N}(3,1)$
2	1000	$y \sim 0.9\,\mathcal{N}(-5,1) + 0.1\,\mathcal{N}(5,1)$
3	200	$y \sim 0.3\,\mathcal{N}(-2,0.8^2) + 0.3\,\mathcal{N}(0,0.8^2) + 0.4\,\mathcal{N}(2,1)$
4	400	$y \sim 0.5\,t_5(-5,1) + 0.5\,\mathsf{SkewNormal}(5,1,2)$
5,6	400	$y \sim 0.5\,\mathcal{N}(-3\cdot 1_d, I_d) + 0.5\,\mathcal{N}(3\cdot 1_d, I_d) \quad d = 2,5$

Python tests: parameters

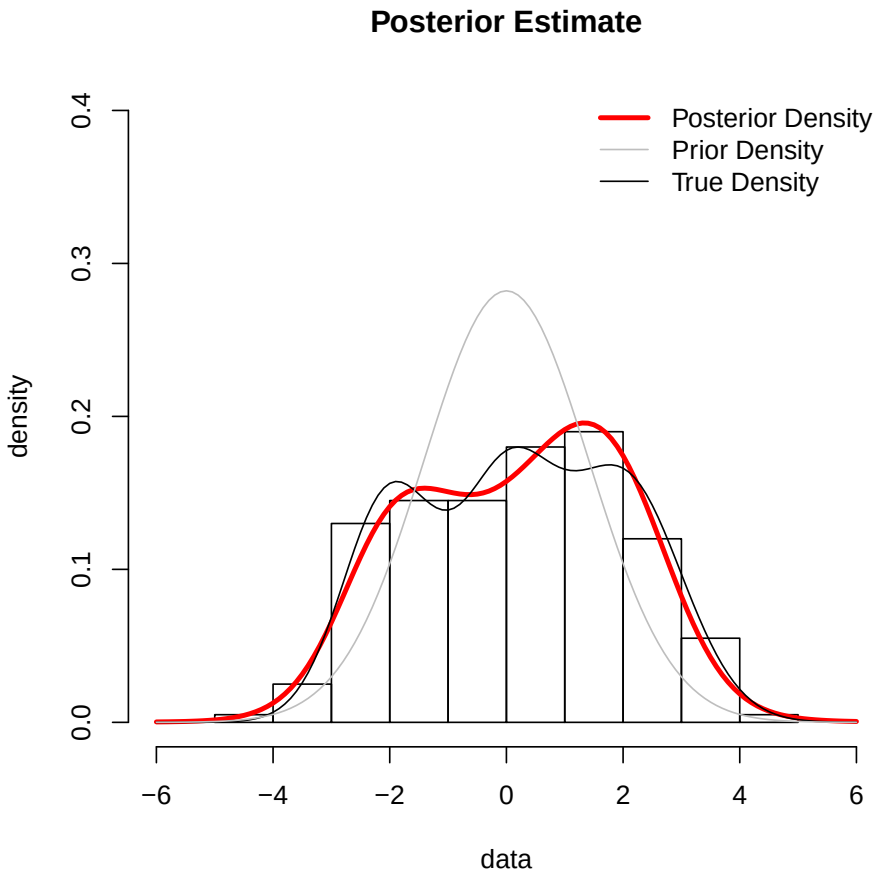
- Hyperparameters for all NNIG hierarchies:
  - $\mu_0 = 0.0$
  - $\lambda = 0.1$
  - $\alpha = 2$
  - $\beta = 2$
- Hyperparameters for all d-dimensional NNW hierarchies:
  - $\mu_0 = \overline{y}$
  - $\lambda = 0.2$
  - $\nu = d + 3$
  - $\tau_0 = \frac{1}{\nu} I_d$
- Dirichlet Mixture with $M = 1$
- Method parameters for `Neal2`:
  - `maxiter` $= 500$
  - `burnin` $= 100$



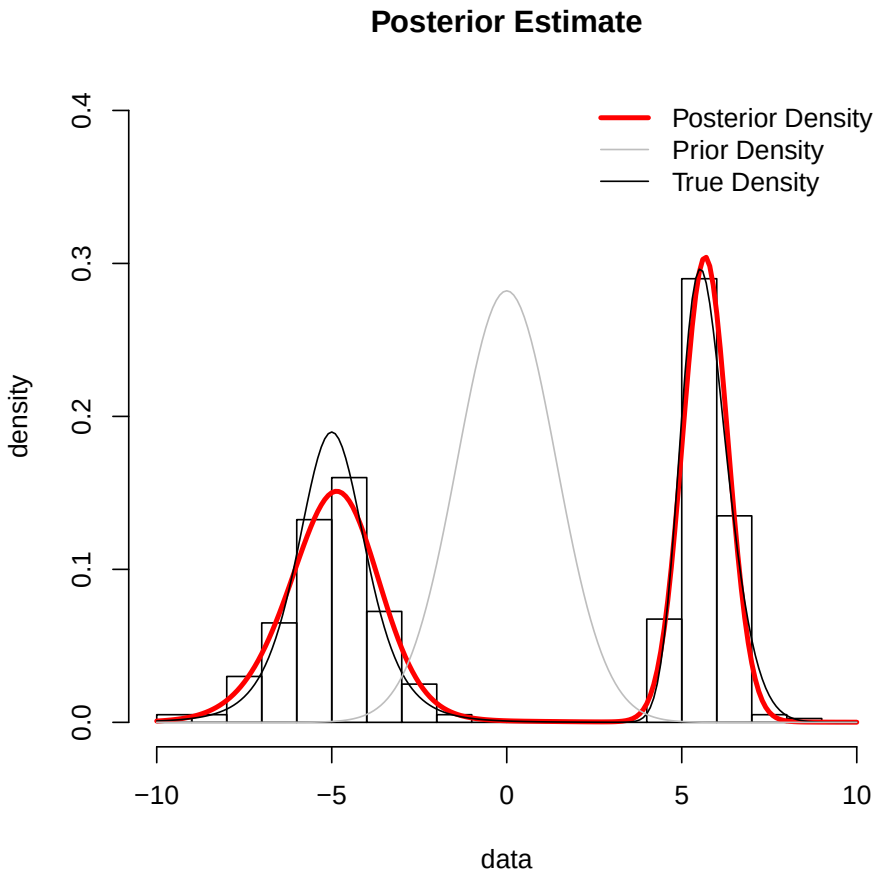
Test 1. $ARI = 1.0$



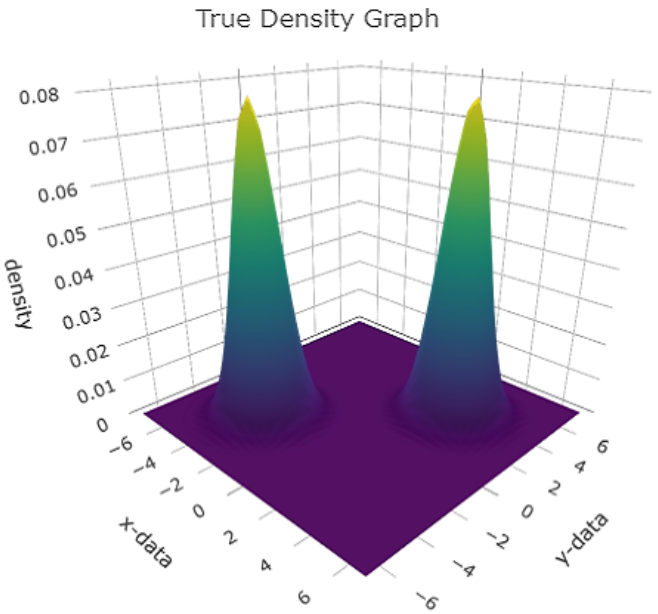
Test 2. $ARI = 1.0$

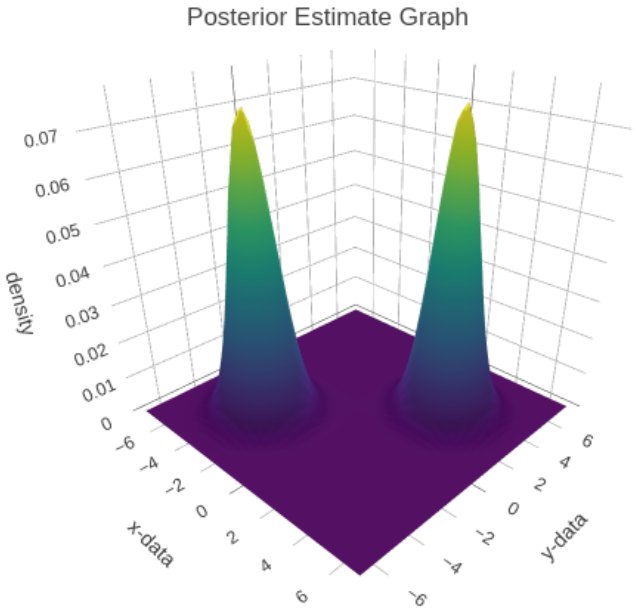


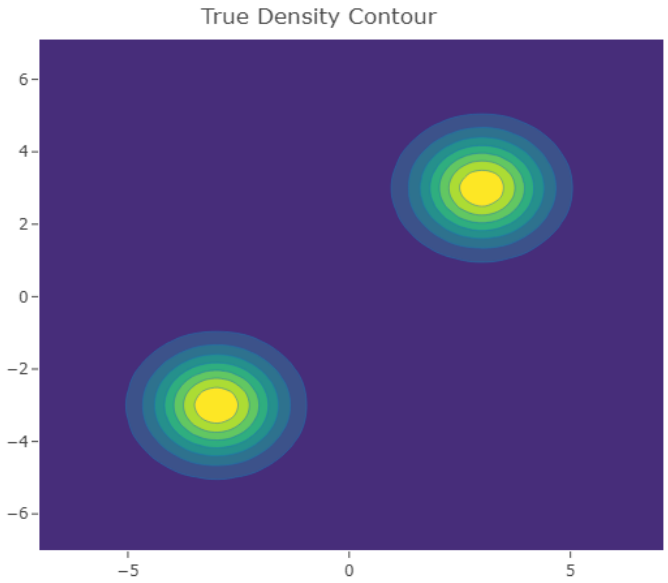
Test 3. $ARI = 0.45$

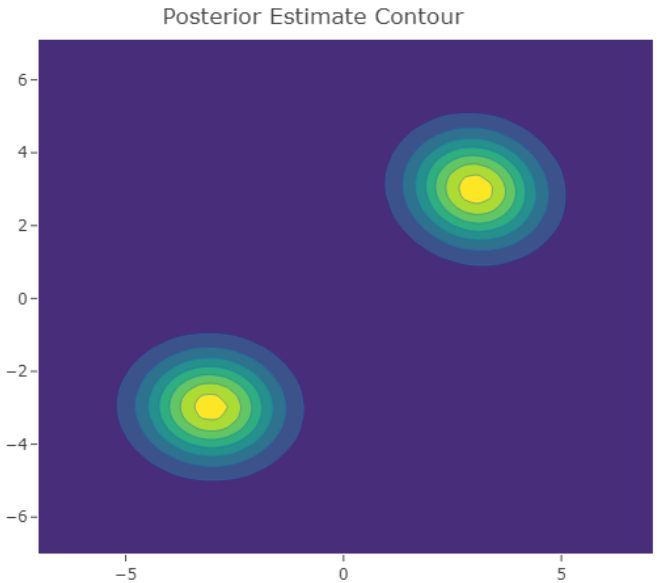


Test 4. $ARI = 0.99$



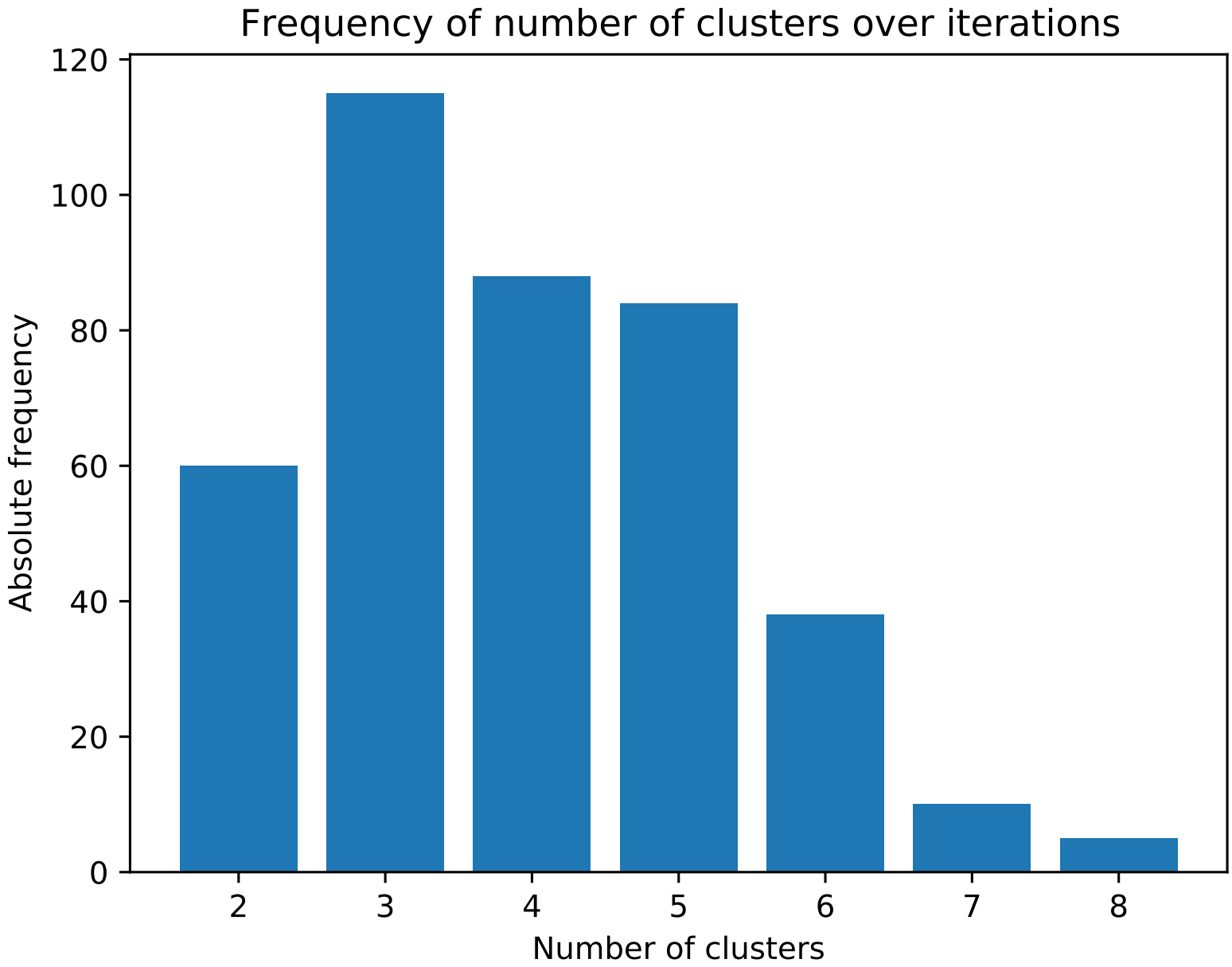






Test 5. True pdf on the left, posterior pdf on the right. $ARI = 1.0$

Example of `chain_barplot()` for test 1



Extensions

- Complete runtime implementation
- Adaptation for hyper-priors on total mass M or on `Hypers`
- Implementation of other `Hierarchy` / `Mixture` / `Algorithm` classes (split-and-merge, slice samplers, handling non-conjugacy)
- Interface with R
- FileCollector without `get_chain()`
- Code parallelization (in `cluster_estimate()`; consensus Monte Carlo)
- Integration of generic Normalized Random Measures
- Full generalization?

Bibliography


P. Muller, F. A. Quintana, *Bayesian Nonparametric Data Analysis*


R. M. Neal (2000), *Markov Chain Sampling Methods for Dirichlet Process Mixture Models*


H. Ishwaran, L. F. James (2001), *Gibbs Sampling Methods for Stick-Breaking Priors*


J. Pitman, M. Yor (1997), *The two-parameter Poisson-Dirichlet distribution derived from a stable subordinator*


K. P. Murphy (2007), *Conjugate Bayesian analysis of the Gaussian distribution*


Rand W. (1971), *Objective Criteria for the Evaluation of Clustering Methods*


Stan documentation: `http://mc-stan.org/math`. Code found at `https://github.com/stan-dev/math`


Eigen documentation: `https://eigen.tuxfamily.org/dox`. Code is included in the Stan package


Protocol Buffers Tutorial for C++: `https://developers.google.com/protocol-buffers/docs/cpptutorial`. Code found at `https://github.com/protocolbuffers/protobuf`


pybind11 tutorial and documentation: `http://pybind11.readthedocs.org/en/master`


scikit-learn user guide: `https://scikit-learn.org/stable/user_guide.html`


Codes of Mario Beraha and Riccardo Corradin for similar projects, found at `https://github.com/mberaha/partial_exchangeability` and `https://github.com/rcorradin/BNPmix` respectively


Course material for Bayesian Statistics by Prof. Guglielmi: `https://beep.metid.polimi.it/web/2019-20-bayesian-statistics-alessandra-guglielmi-/`


Course material for Advanced Programming for Scientific Computing by Prof. Formaggia: `https://beep.metid.polimi.it/web/102725282`