February 11, 2020

Abstract

1

2 Introduction

This report presents the development of a C++ Bayesian Non parametric library containing Marcov Chain sampling methods for density estimation and clustering. In a Bayesian Non-parametric setting we focused on the Dirichlet process (DP) and its extensions, one of the most widely used priors due to its flexibility and computational ease.

3 Dirichlet Process

Formal definition : Let $M > 0$ and G_0 be a probability measure defined on S. A DP with parameters (M, G_0) is a random probability measure G defined on S which assigns probability $G(B)$ to every (measurable) set B such that for each (measurable) finite partition $B_1, ..., B_k$ of S, the joint distribution of the vector $(G(B_1), ..., G(B_k))$ is the Dirichlet distribution with parameters

$$(MG_0(B_1), ..., MG_0(B_k)). \tag{1}$$

The parameter M is called the precision or total mass parameter, G_0 is the centering measure, and the product MG_0 is referred to as the base measure of the DP.

The basic DP model has the form:

$$y_i | G \overset{\text{iid}}{\sim} G, \quad i = 1, \ldots, n$$
$$G \sim DP(MG_0)$$

A key property is that the DP is conjugate with respect to i.i.d sampling so that the posterior base distribution is a weighted average of the prior base distribution G_0 and the empirical distribution of the data, with the weighting controlled by M :

$$G | \mathbf{y} \sim DP(MG_0 + \sum_{i=1}^{n} \delta_{y_i}). \tag{2}$$

And the marginal distribution will be the result of the product of the conditionals:

$$p(y_i | y_1, ..., y_{i-1}) = \frac{1}{M + i - 1} \sum_{h=1}^{n-1} \delta_{y_h}(y_i) + \frac{M}{M + i - 1} G_0(y_i). \tag{3}$$

An important property of the DP is the discrete nature of G. As a discrete random probability measure we can always write G as a weighted sum of point masses. A useful property based on the discrete nature of the process is his stick-breaking representation, i.e. G can be written as:

$$G(\cdot) = \sum_{k=1}^{+\infty} w_k \delta_{m_k}(\cdot) \tag{4}$$

with $m_k \stackrel{\text{iid}}{\sim} G_0$ and the random weights constructed as $w_k = v_k \prod_{l<k} (1 - v_l)$ where v_k are independent Be(1,M)random variables.

In many applications in which we are interested in a continuous density estimation this discreteness can represents a limit. It's common choice to use a Dirichlet Process Mixture (DPM) model where the DP random measure is the mixing measure for the parameters of a parametric continuous kernel function.

4 Dirichlet Process Mixture Model

Extending the DP by convolving G with a kernel F, the model will have the form:

$$y_i | G \sim F_G(y) = \int F(y, \vartheta) \, G(\mathrm{d}\vartheta), \quad i = 1, \dots, n$$
$$G \sim DP(MG_0)$$

An equivalent hierarchical model is:

$$y_i | \vartheta_i \stackrel{\perp\!\!\!\perp}{\sim} F(\cdot, \vartheta_i), \quad i = 1, \dots, n$$
$$\vartheta_i | G \stackrel{\text{iid}}{\sim} G, \quad i = 1, \dots, n$$
$$G \sim DP(MG_0)$$

where the *latent variables* ϑ_i are introduced, one per unit. Since G is discrete, we know that two independent draws ϑ_i and ϑ_j from G can be equal with positive probability. In this way the DPM model induces a probability model on clusters and an object of interest starting from this model is the partitioning induced by the clustering as well as the density estimation.

Considering n data units, each ϑ_i will have one of the k unique values ϕ_j. An estimation of the number of the unique values is $M \log(n) \ll n$. Calling c_i the *allocation* parameters to the clusters such that $c_i = j$ if $\vartheta_i = \phi_j$ the model can be thought as the limit as K goes to infinity of finite mixture model with K components:

$$(Y_i | \phi, c_i) \sim F(\cdot, \phi_{c_i})$$
$$(c_i | \mathbf{p}) \sim \sum_{k=1}^{K} p_k \delta_k(\cdot)$$
$$\phi_c \sim G_0$$
$$\mathbf{p} \sim \mathrm{Dir}(M/K, \dots, M/K)$$

where $(p_1, ..., p_K)$ represent the mixing proportions for the classes and each theta is defined by the latent class c and the corresponding parameters ϕ_c.

4.1 Normal Normal-InverseGamma Model

A very common choice is the Gaussian Mixture Model, opting for a Normal kernel and as base measure G_0 the conjugate Normal-InverseGamma:

$$k(y|\vartheta) = N(y, \mu, \sigma^2)$$

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

Thanks to the conjugacy the predictive distribution for a new observation $\widetilde{y}$ can be found analytically:

$$p(\widetilde{y}|\mu_0, \lambda_0, \alpha_0, \beta_0) = \int k(\widetilde{y}|\vartheta)p(\vartheta, G_0)d\vartheta = \frac{1}{\widetilde{\sigma}}\text{t-Student}\left(\frac{\widetilde{y} - \widetilde{\mu}}{\widetilde{\sigma}}, |\widetilde{v}\right) \quad (5)$$

where $\widetilde{v} = 2\alpha_0$, $\widetilde{\mu} = \mu_0$ and $\widetilde{\sigma} = \sqrt{\frac{\beta_0(\lambda_0 + 1)}{\alpha_0 \lambda_0}}$

The posterior distribution is again a Normal-InvGamma:

$$p(\vartheta|y, \mu_0, \lambda_0, \alpha_0, \beta_0) = N\left(\mu|\mu_n, \frac{\sigma^2}{\lambda_0 + n}\right) \text{Inv-Gamma}(\sigma^2|\alpha_n, \beta_n) \quad (6)$$

with $\mu_n = \frac{\lambda_0 \mu_0 \bar{y} + n}{\lambda_0 + n}$, $\alpha_n = \alpha_0 + \frac{n}{2}$ and $\beta_n = \beta_0 + \frac{1}{2}\sum_{i=1}^n (y_i - \bar{y})^2 + \frac{\lambda_0 n(\bar{y} - \mu_0)^2}{2(\lambda_0 + n)}$

5 Methods

Starting from the hierarchical model (n) a direct approach is simply drawing values for each ϑ_i from its conditional given the data and the other ϑ_j, but as we discussed before there is an high probability for ties among them and this can result in a slow convergence.

5.1 Neal2

To solve and make it more efficient Neal proposed, starting from the discrete model (n), a Gibbs sampling method, integrating out the mixing proportion **p**. Assuming the current state of Markov chain consists of $(c_1, ..., c_n)$ and the component parameters ϕ_c for all c, the Gibbs sampler consists of drawing values for each c_i given the conditional probabilities:

$$\mathbb{P}(c_i = c|\mathbf{c}_{-i}, y_i, \boldsymbol{\phi}) \propto \frac{n_{-i,c} + M/K}{n - 1 + M} F(y_i, \phi_c)$$

and consequently a new value for each ϕ_c given the data belonging to that class. The passage to the infinite case is done taking the limit of K to infinity in the conditional distribution of c_i that becomes:

$$\mathbb{P}(c_i = c|\mathbf{c}_{-i}, y_i, \boldsymbol{\phi}) \propto \frac{n_{-i,c}}{n - 1 + M} F(y_i, \phi_c)$$

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

and considering only the ϕ_c associated with some observation, keeping feasible the sampling. At this point the algorithm works iteratively sampling c and ϕ. For each observation i c_i is updated according to its conditional distribution. It can be set either to one of the other components currently associated with some observation or to a new mixture component. If the new value of c_i is different from all the other c_j a value for ϕ_{ci} is drawn from the posterior distribution H_i, based on the prior G_0 and the single observation y_i. Then for all the classes the sample for ϕ_c is done considering the posterior distribution based on the prior and all the observations belonging to the specific class.

The probability of setting c_i to a new component involves the integral $\int F(y_i, \phi) \, G_0(\mathrm{d}\phi)$, which is difficult in the non-conjugate case, as well as the sample from H_i.

5.2   Neal8

To handle non-conjugate priors Neal proposed a second Markov chain sampling procedure where the state is extended by the addition of auxiliary parameters. This technique allows to update the c_i avoiding the integration with respect to G_0.

In this case the prior for c_i are:

$$\text{If } c = c_j \text{ for some } j: \; \mathbb{P}(c_i = c | \mathbf{c}_{-i}) = \frac{n_{-i,c}}{n - 1 + M}$$

$$\mathbb{P}(c_i \neq c_j \text{ for all } j) = \frac{M}{n - 1 + M}$$

where the probability of selecting a new component is split among the m auxiliary components. Maintaining the same structure as the *Algorithm* 2, the *Algorithm* 8 is composed of two steps, where the components of the Markov Chain state (c, ϕ) are repeatedly sampled. The first step scans all the observations and evaluates each c_i. If this is equal to another c_j then all the auxiliary variables are drawn from G_0. If it is a singleton then it is linked to one of the auxiliary variable with the corresponding value of ϕ_c while the others are drawn as before from G_0. Then c_i is updated according to the conditional probabilities:

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

Once all the ϕ_c not associated anymore with any observation are discarded, the algorithm proceeds with the sampling for ϕ_c for all the classes.

5.3   Blocked Gibbs

6 Implementation

As far as code implementation goes, the aforementioned algorithms all share
the following structure:

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


void run(){
    initialize();
    unsigned int iter = 0;
    while(iter < maxiter){
        step();
        if(iter >= burnin){
            save_iteration(iter);
        }
        iter++;
    }
}
```

In particular, the Neal2 and Neal8 algorithms do not have a `sample_weights()`
phase. Each implemented algorithm will be discussed in detail in its own section.
As for the general structure of an algorithm class, a template approach was
chosen, to allow the use of several layers of complexity based on the needs of
the user:

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

That is, `Hierarchy<>`, `Hypers`, and `Mixture` are not actual implemented classes,
but rather proxy names for classes which will be received as *parameters* by the
algorithm class. These classes must have a *common interface* in order for them
to be passed as parameters, as explained in the following section. An example
with actual class names (as it would be called from the `main.cpp`) is:

```
Neal8<NNIGHierarchy, HypersFixed, SimpleMixture> sampler8;
```

6.1 Auxiliary classes

- The `Mixture` classes contain all information about the mixing part of
  the DPM model, namely the total mass parameter and its prior distribu-
  tion, if any. We implemented the `SimpleMixture` class, which represents
  a fixed total mass parameter without any prior on it, and contains the
  `totalmass` member as well as a getter and a setter (`get_totalmass()`,
  `set_totalmass()`).

- The `Hypers` classes contain all information about the hyperparameters of
  the hierarchy, including their values (if fixed) or their prior distributions
  (if not). We implemented the `HypersFixedNNIG` class, which contains the

four fixed parameters `mu0`, `lambda`, `alpha0`, and `beta0` of the Normal-NIG hierarchical model, and their respective getters and setters.

- The `Hierarchy<Hypers>` classes are template classes themselves and accept any `Hypers` class as template parameter. A `Hierarchy<>` class contains a vector `state` which stores the current values of the likelihood parameters, as well as a pointer to a `Hypers` object – this is why `Hypers` is required as a parameter for `Hierarchy<>`. A pointer is chosen instead of an actual object, since multiple `Hierarchy<>` objects will be created and stored by the algorithms; the `state`s of these objects will of course share the same prior, and with a pointer to `Hypers` the updating of the prior will only happen once rather than one time per object. A `Hierarchy<>` class also contains functions to:

  - evaluate the marginal distribution (provided it is known in closed form) and the log-likelihood in a given set of points, given the current `state`;
  - compute the posterior parameters with respect to a given set of data points;
  - generate new values for the `state` both according to its prior and to its posterior distribution;
  - get and set class members, as with the other classes.

  In particular, we implemented the `HierarchyNNIG` class, which represents the Normal-NIG model described earlier.

Any class representing any type of hierarchy or parameters can be built as long as it possesses the above interface, which is required for their use in the implemented algorithms.

We will be now first examining the `Neal8` class as an example.

6.2 `Neal8` algorithm

Aside from the usual getters and setters, as well as constructors, the `Neal8` class contains the following members:

```
unsigned int n_aux;
unsigned int maxiter;
unsigned int burnin;
unsigned int num_clusters;
std::mt19937 rng; // random number generating engine
```

These are the parameters of the method, and are rather self-explanatory. Their values are initialized either via the constructors or the setters. If `num_clusters` is not provided, it will be automatically set equal to the number of data points, thus starting the algorithm with one datum per cluster.

The data and values containers are as follows:

```
std::vector<double> data;
std::vector<unsigned int> allocations;
std::vector<Hierarchy<Hypers>> unique_values;
std::vector<Hierarchy<Hypers>> aux_unique_values;
Mixture mixture;
```

The algorithm will keep track of the labels representing assignments to clusters via the `allocations` vector. For instance, if one has `allocations[5] = 2`, it means that datum number 5 is associated to cluster number 2. Note that indexing for both data and clusters starts at zero, so this actually means that we have the sixth datum being assigned to the third cluster.

The containers for the unique values ϕ hold objects of type `Hierarchy<>` because each ϕ is associated to a cluster, which is in fact a small hierarchy that can have its own hyperprior in the general case. The same reasoning goes for `aux_unique_values`, which will be referred to as the vector of *auxiliary blocks*, from which the algorithm may draw in order to generate new clusters.

As for the members used for running the algorithm:

```
void initialize();
void sample_allocations();
void sample_unique_values();
void step(){
    sample_allocations();
    sample_unique_values();
}
void save_iteration(unsigned int iter);
void run();
```

Aside from `run()`, whose code was shown at the beginning of this section, we shall briefly describe the implementation of these functions:

- `initialize()` creates `num_clusters` clusters and randomly assigns each datum to one of them, while making sure that each cluster contains at least one. This assignment is done through changing `allocations` components, as explained earlier.

- In `sample_allocations()`, a loop is performed over all data points $i = 1 : n$. A vector `card` is first filled, with `card[j]` being the cardinality of cluster j. The algorithm mandates that `data[i]` be moved to another cluster; thus, if the current cluster is a singleton, its ϕ values are transferred to the first auxiliary block. Then, each auxiliary block (except the first one if the above case occurred) generates new ϕ values via the hierarchy's `draw()` function. Now, a new cluster, that is a new ϕ value, for `data[i]` needs to be drawn. A vector `probas` with `n_unique+n_aux` components is filled with the probabilities of each ϕ being extracted. Computations involve, among other things, the `card` vector, the `log_like()` evaluated in `data[i]`, and the total mass parameter, as shown in the previous sections. Then, the new value for `allocations[i]` is randomly drawn according to the computed `probas`. Finally, four different cases of updating `unique_values` and `card` are handled separately, depending on whether the old cluster was a singleton or not, and whether an auxiliary block or an already existing cluster was chosen as the new cluster for `data[i]`. This is done because depending on the case, clusters are either unchanged, increased by one, decreased by one, or moved around.

- In `sample_unique_values()`, for each cluster j, their ϕ values are updated through the `sample_given_data()` function, which takes as argument the vector `curr_data` of data points which belong to cluster j. Since we only

keep track of clusters via their labels in `allocations`, we do not have a vector of actual data points stored for each cluster. Thus we must fill, before the loop on j, a matrix `clust_idxs` whose column k contains the index of data points belonging to cluster k. `clust_idxs` is then used in the j loop to fill `curr_data` with the actual data points of cluster j.

- `save_iteration` will be examined in a later section.

6.3 `Neal2` algorithm

The structure of the `Neal2` class is similar to the one of `Neal8` described above. The only relevant differences are the obvious lack of `aux_unique_values` and most of the `sample_allocations()` phase. Since this algorithm exploits conjugacy, this function must be specifically implemented for each different hierarchy. In our case, a Normal-NIG specialization for the `Neal2` template class was implemented. In `sample_allocations()`, a loop is performed over data points i `card` vector is built, just as before. The `probas` vector of weights for the new allocation value is computed by also using the marginal density in `data[i]`, which is known to be a Student's t as mentioned in the appropriate section. After the new `allocations[i]` is drawn according to `probas`, four cases are handled separately as before, depending on whether the old cluster was a singleton and whether `data[i]` is assigned to a new cluster. Indeed, in such a case, a new ϕ value for it must be generated, and this must be handled differently by the code if an old singleton cluster was just destroyed (as the new cluster must take its former place).

7 Applications

These algorithms can also be used for two useful practical purposes: *cluster estimation* and *density estimation*. Both processes, however, require the whole chain to be saved, that is, at each iteration the current values of states and allocations must be stored in some data structure. For this purpose, we used the Protocol Buffers library, which needs a short introduction.

7.1 Storing values with `protobuf`

Protocol Buffers, or `protobuf` for short, was developed by Google and allows automatic generation of data-storing C++ classes by defining a class skeleton in a `.proto` file. This also allows easy interfacing with other programming languages such as R and Python.
We built our template as follows:

```
message UniqueValues {
    repeated double params = 1;
}
message IterationOutput {
    repeated int32 allocations = 1;
    repeated UniqueValues phi = 2;
}
message ChainOutput {
```

```
    repeated IterationOutput state = 1;
}
```

Here `message` and `repeated` are the `protobuf` equivalent of classes and vectors respectively, while the numbers 1 and 2 just act as identifiers for the fields in the messages. After generating the corresponding C++ classes via the `protoc` compiler, we were able to add the following members to the `Neal8` and `Neal2` classes:

```
ChainOutput chain;
IterationOutput best_clust;
std::pair< std::vector<double>, Eigen::VectorXd > density;
```

For each iteration after the burn-in phase, the `save_iteration()` function saves all state values of the current iteration into the `chain` pseudo-vector in the appropriate structure. On the other hand, `best_clust` represents the state of a single iteration, and it is the object where the result of the clustering analysis will be saved. The `density` object shares a similar purpose for the density estimation part, albeit not actually generated via `protobuf`. It will be filled with a grid of points in which the density will be evaluated, and the evaluations of the density themselves.

We will be explaining in thorough detail these two useful applications in the next lines.

7.2   Cluster estimation

Suppose we wish to estimate the real clustering of the data, assuming the DPM model holds true. A first rough estimate is the *final clustering*, that is, the state values corresponding to the last iteration of the algorithm. This estimate does not require an appropriate function to be implemented, since the state values are already available in `allocations` and `unique_values` after the algorithm is `run()`. However, due to oscillating behavior, the last clustering may not be the optimal one. Instead, we chose to implement a *least square* estimate in the following function:

```
unsigned int cluster_estimate();
```

This function exploits the `chain` pseudo-vector, in which states of all iterations of the algorithm were saved via `save_iteration()` (of course, only after the burn-in phase) and the `protobuf` library. This function loops over all `IterationOutput` objects in `chain`, finds the iteration at which the best clustering occurred, saves the whole object into the `best_clust` class member, and returns the iteration number of this best clustering. As briefly touched upon earlier, the best clustering is found via the minimization of the squared posterior *Binder's loss function*. An equivalent approach is computing the so-called *dissimilarity matrix* for each iteration, computing its sample mean over all iterations, and finding the iteration that is the closest to the mean with respect to the *Frobenius norm*. More specifically, for each iteration k, the dissimilarity matrix $D^{(k)}$ is a symmetric, binary n-by-n matrix (where n is the number of available data points) whose entries $D_{ij}^{(k)}$ are 1 if datum i and j are placed in the same cluster at iteration k and 0 otherwise. After each $D^{(k)}$ and the sample mean $\bar{D} = \frac{1}{K} \sum_k D^{(k)}$ are computed, where K is the number of iterations

(not counting the ones in the burn-in phase), the best clustering $\hat{k}$ is found by minimizing the Frobenius norm of the difference with $\bar{D}$:

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

By virtue of the involved matrices being symmetric, the latter summation can be computed over all $i < j$ instead of all i, j for efficiency.

7.3 Density estimation

One other important application of clustering algorithms is the estimation of the density according to which the data points are distributed. This is done differently in both the `Neal2` and `Neal8` algorithms, as the former can exploit the conjugacy of the hierarchical model. In either case, the following function was implemented:

```
void eval_density(const std::vector<double> grid);
```

It accepts a grid of points in which the density will be evaluated. This grid is stored in the `density` member object, as well as the computed evaluations themselves in form of a vector from the `Eigen` library. Just like for the cluster estimate, the computation will access all iterations stored in the `chain` pseudo-vector. In both `Neal8` and `Neal2`, a loop is performed over the iterations k. Suppose this iterations has J clusters, that is, $j = 0 : J - 1$. The `card` vector is once again computed, where `card[j]` $= n_j^{(k)}$ is the cardinality of cluster j. Then, for each point x in `grid`, we compute the local estimate of the density, that is, only taking iteration k into account:

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

That is, the local estimate is a weighted mean of the likelihood given the unique values $\phi_j^{(k)}$ of cluster j and the marginal distribution $m(x)$, taken from the appropriate function in the `Hierarchy<>` class. The weights of the clusters are proportional to their size $n_j^{(k)}$, while the "virtual" cluster of the marginal counts as having size M, the total mass parameter (n is instead the number of data points, as per usual). The marginal distribution is only known under the conjugacy assumption in the `Neal2` algorithm. In particular, for a Normal-NIG model $m(x)$ is a Student's t as explained in section ?. In the `Neal8` algorithm, $m(x)$ is not available in closed form, and thus it is replaced in the above formula by the following approximation:

$$\tilde{m}(x) = \sum_{h=0}^{m-1} \frac{M}{m} F\left(x | \phi_h \right)$$

where we use m unique values, that is, one for each of the $m = $ `n_aux` auxiliary blocks of the algorithm, drawn from the base measure: $\phi_h \overset{\text{iid}}{\sim} G_0, \; h = 0 : m - 1$.

Finally, the *empirical density* is computed as the mean over all iterations:

$$f(x) = \frac{1}{K} \sum_k f^{(k)}(x)$$

and saved into the `density` object.

7.4 Saving estimates to files

We also implemented the following functions in each `Algorithm` class, which save data from the class into text files in order to ease exportation to other programs or computers:

```
const void write_final_clustering_to_file(
        std::string filename = "final_clust.csv");
const void write_best_clustering_to_file(
        std::string filename = "best_clust.csv");
const void write_chain_to_file(
        std::string filename = "chain.csv");
const void write_density_to_file(
        std::string filename = "density.csv");
```

They can be called as need be from the `main.cpp`. If a file name is not provided, the above default names will be used.

8 Results

TBD

9 Extensions

The `bnplib` library has several possible extensions:

- New types of `Hypers` classes can be implemented, for example ones containing hyper-priors for some of the parameters if the model. The algorithm must be modified accordingly, for instance by adding extra steps (which are skipped if the hyperparameters are fixed). Changes depend on the type of parameter for which a prior is used; for example, a prior on the total mass parameter involves different steps than a prior on the parameters of the base measure. For a general outline of the necessary changes, see [2] section 7.

- Hierarchies other than the Normal-NIG can be created. This is enough to run `Neal8` by passing the class name as parameter, but other algorithms such as the conjugacy-dependent `Neal2` may need an hoc implementation of some segments of code.

- Interfaces with both R and Python can be easily implemented, thanks to the data structures provided by `protobuf` and the already-available libraries `Rcpp` and `pybind` respectively.

- Algorithms can be re-adapted for the use of other mixture models, such as the Pitman-Yor process.

- Conjugacy-dependent algorithms such as `Neal2` can be re-adapted to account for non-conjugacy, for example by using an Hamiltonian Monte Carlo sampler.

- Finally, a *full generalization* of the library might be possible. That is, given the distributions of the likelihood, hyperparameters, etc, one might want an algorithm that works for the chosen specific model without needing and explicit implementation for it. This means, among other things, that one has to handle non-conjugacy for the general case. The main issue is that Stan distribution functions do not accept vectors of parameter values as arguments; thus, the updated values for distributions must be explicitly enumerated and given as arguments one by one to the Stan function. This requires to know in advance the number of parameters for all such distributions, which is impossible in the general case. Some advanced C++ techniques may be used to circumvent this hindrance, such as argument unpackers that transform a vector into a list of function arguments, and variadic templates, which are templates that accept any number of arguments. Theoretically, the latter would also allow the use of priors on the parameters of the hyper-prior itself, and so on, adding layers of uncertainty ad libitum. Of course this is a daunting task that is far beyond the scope of this library, although we do think it is possible to achieve with reasonable effort.

References

[1] Muller, Quintana, *Bayesian Nonparametric Data Analysis*

[2] Neal (2000), *Markov Chain Sampling Methods for Dirichlet Process Mixture Models*

[3] Ishwaran, James (2001), *Gibbs Sampling Methods for Stick-Breaking Priors*

[4] Ross, Markwick, *dirichletprocess: An R Package for Fitting Complex Bayesian Nonparametric Models*, found at `https://cran.r-project.org/package=dirichletprocess`

[5] Stan documentation: `http://mc-stan.org/math`. Code found at `https://github.com/stan-dev/math` (version 3.0.0 was used) and it also includes other needed libraries: Boost, Eigen, SUNDIALS, Intel TBB

[6] Eigen documentation: `https://eigen.tuxfamily.org/dox`.
Code is included in the Stan package (version 3.3.3 is used there)

[7] Protocol Buffers Tutorial for C++: `https://developers.google.com/protocol-buffers/docs/cpptutorial`.
Code found at `https://github.com/protocolbuffers/protobuf` (version 3.11.0 was used)