`bnplib`: A Nonparametric C++ Library

Bruno Guindani
Elena Zazzetti

Professors: Alessandra Guglielmi, Luca Formaggia
Advisor: Mario Beraha

Politecnico di Milano

June 12th, 2020

https://github.com/poliprojects/BNPlib

Abstract

We present a C++ library that exploits a Bayesian Non-Parametric setting in order to conduct unidimensional and multidimensional data analysis. In such a setting, our main goals are density estimation and clustering analysis. Several iterative algorithms are available that make use of Gibbs sampling, building a Markov chain that reaches convergence at a reasonably fast pace. In particular, we focused on implementing some algorithms introduced by Neal in 2000. After running one of these algorithms, density and cluster estimation can then be conducted by using the provided auxiliary tools. In this report, after an overview of the underlying Bayesian model and a roundup of the algorithms' state of the art, we delve into the details of our implementation and then present an example of data analysis, whilst providing the theoretical background for our estimates. We also include a description for the implemented Python interface for this C++ library.

Contents

Chapter 1

Introduction

This report presents the development of a C++ library containing Markov chain sampling algorithms for two major goals: *density estimation* and *clustering analysis* of a given set of data points. In a Bayesian nonparametric (BNP) setting, we focused on the Dirichlet process, one of the most widely used priors due to its flexibility and computational ease, and its extensions and generalizations. Hereafter, we will assume that the underlying model for the given data points is a Dirichlet process mixture model, which is an enhancement of the simpler Dirichlet model. (For a more detailed discussion of the nonparametric models, as well as references for all theoretical details included in this section, see [1] chapter 1 and 2.)

1.1 A quick introduction to Bayesian statistics

Bayesian statistics is a branch of mathematics with goals similar to regular statistics, which also holds the more accurate name of *frequentist statistics*. In both theories, data points y are considered as realizations of random variables, often iid – independent and identically distributed, with their distribution having one or more fixed and unknown parameters ϑ, such as mean and variance in the Gaussian cases. However, the frequentist approach is heavily focused on data and on pure information that can be extracted from it, for instance via estimates based on some form of sample mean. By contrast, the Bayesian approach brings the data scientist's prior knowledge about the data into the picture; such knowledge is assumed to be approximately true, and the data is used to give it refinements and updates. In a formal Bayesian setting, this prior knowledge takes on the form of a distribution $p(\vartheta)$ on the parameters of the data, called *prior distribution*, or prior for short. This is a crucial difference with respect to frequentist statistics, where parameters ϑ are unknown but assumed to be fixed, whilst in the Bayesian environment they are treated as *random variables* for all intents and purposes. This explains a major advantage of the Bayesian theory: it is naturally suited for non-pointwise estimates, mainly in the form of parameter distributions or their summary statistics, and therefore these are much easier to obtain than in the frequentist counterpart. After taking the given data points into consideration, the parameter estimate provided by the prior is updated into the so-called *posterior distribution* $p(\vartheta|y)$, or posterior for

short, for the parameters. The conditioning symbol indicates that the actual values of the realizations are used for a new better estimate of the parameters. Similarly, since data are now distributed according to the random ϑ, we use the notation $f(y|\vartheta)$ for the data distribution, which in this context is called *likelihood*.
One can easily see that Bayesian statistics makes heavy use of conditional probabilities; so much so, in fact, that its very name is based off of the generalization of a well-known result for conditional probabilities, the *Bayes theorem*. In particular, this theorem states that, given $y_1, \dots, y_n$ iid random variables with joint likelihood $f(y|\vartheta) = f(y_1, \dots, y_n|\vartheta)$ and parameters with prior $p(\vartheta)$, the posterior for ϑ is given by

$$p(\vartheta|y) = \frac{p(\vartheta)f(y|\vartheta)}{\int p(\vartheta)f(y|\vartheta)\,\mathrm{d}y} = \frac{p(\vartheta)f(y|\vartheta)}{m(y)} \propto p(\vartheta)f(y|\vartheta).$$

The denominator $m(y) = \int p(\vartheta)f(y|\vartheta)\,\mathrm{d}y$ is the *marginal distribution* of data y, that is, its overall distribution without the knowledge of its parameters ϑ. It is often treated as an unimportant normalization constant, since it does not contain ϑ, and therefore one often uses the last equality, which highlights the posterior's dependence on both the prior and the likelihood.
A coveted property for a Bayesian model is *conjugacy*, that is to say that both the prior and the posterior distribution have the same form, e.g. they are both Gaussian distributions (most likely their parameters would both be different). This property, or lack thereof, can make the difference between a posterior distribution being extremely easy to compute, and being flat out impossible to compute in closed form. We shall see some examples of conjugate models later on in this section.
Finally, note that a Bayesian model may still employ frequentist tools to better incorporate data information into the prior; the most common example of this is setting the mean of the prior distribution as the sample mean of the given data.

1.1.1 Extended models

One is also allowed to use a more layered model, in which the parameters of the prior distribution, called *hyperparameters*, also have prior distributions on them – these are called *hyperpriors*. The result is as follows:

$$\begin{aligned} y_1, \dots, y_n|\vartheta, \lambda &\overset{\text{iid}}{\sim} f(y|\vartheta, \lambda) \\ \vartheta|\lambda &\sim p(\vartheta|\lambda) \\ \lambda &\sim \Pi(\lambda) \end{aligned}$$

In fact, one can add as many layers as needed, adding priors to other priors' parameters, although one hyperprior like in the above model is generally considered enough to handle the complexity of most problems. These are called *hierarchical models*.
Another kind of advanced Bayesian structure is the so-called *nonparametric model*, in which the entire likelihood is assumed to be random. This means that there are infinitely many points which are randomly generated, that is, we have an infinite-dimensional parameter – with a prior distribution for it, of course:

such a likelihood is an example of *random probability measure*.
In the next pages, we will see some examples of elements composing these models, which can be divided into two classes: the *mixture* component and the *hierarchy* component.

1.2 Dirichlet process model

Let $M > 0$, and let G_0 be a probability measure defined on the state space S. A Dirichlet process with parameters M and G_0, noted as $DP(MG_0)$, is a random probability measure G defined on S which assigns probability $G(B)$ to every set B such that for each finite partition $B_1, \ldots, B_k$ of S, the joint distribution of the vector $(G(B_1), \ldots, G(B_k))$ is the Dirichlet distribution with parameters $(MG_0(B_1), \ldots, MG_0(B_k))$.[1] The value M is called the *total mass* or precision parameter, G_0 is the *centering measure*, and the product MG_0 is the base measure of the DP.
Having observed the iid sample $\{y_1, \ldots, y_n\} \subseteq \mathbb{R}$, the basic DP model takes the following form:

$$\begin{aligned} y_i|G &\overset{\text{iid}}{\sim} G, \quad i = 1, \ldots, n \\ G &\sim DP(MG_0) \end{aligned} \tag{1.1}$$

A key property is that the DP is conjugate with respect to iid 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 weights controlled by M:

$$G|y_1, \ldots, y_n \sim DP\left(MG_0 + \sum_{i=1}^{n} \delta_{y_i}\right). \tag{1.2}$$

Moreover, the marginal distribution for the data will be the product of the sequence of increasing conditionals:

$$p(y_1, \ldots, y_n) = p(y_1) \prod_{i=2}^{n} p(y_i|y_1, \ldots, y_{i-1}),$$

with $y_1 \sim G_0$ and the conditional for $i = 2, 3, \ldots$ being the following:

$$p(y_i|y_1, \ldots, y_{i-1}) = \frac{1}{M + i - 1} \sum_{h=1}^{n-1} \delta_{y_h}(y_i) + \frac{M}{M + i - 1} G_0(y_i). \tag{1.3}$$

The common pattern in both the above expression of the conditional and in the one for the posterior in (1.2) is that the centering measure G_0 is always

[1] The Dirichlet distribution $\text{Dir}(\alpha_1, \ldots, \alpha_k)$ is a k-dimensional generalization of the Bernoulli distribution, given parameters $\alpha_1, \ldots, \alpha_k > 0$. It has support in the $k-1$-dimensional simplex and its probability density function (p.d.f.) is

$$f(\mathbf{x}) = \frac{1}{B(\alpha_1, \ldots, \alpha_k)} \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

with $B(\alpha_1, \ldots, \alpha_k)$ being the k-dimensional Beta function that acts as a normalization constant.

given a weight proportional to M (up to the normalizing constant $M + i - 1$ in the former), whilst for the single points y_i, or the delta distribution centered in them, the weight is 1 for each datum, or k if it has appeared k times. This is the heart of the so-called *Polya's urn* representation model: the probability of a new value being equal to the ones before it is proportional to the number of times this value has appeared in the past, while the probability of the value being generated anew from G_0 is proportional to M. Therefore, the more a value appears, the more likely it is for it to appear again in the future; instead, the total mass acts as the "cardinality" of the action "draw a new value". This is equivalent to having an urn which contains balls of different colors; each time a ball of a certain color is extracted, the ball is placed back into the urn alongside another new ball of the same color. Furthermore, there's a chance proportional to M that instead of extracting a ball from the urn, a ball of a random color, not necessarily one that is already present in the urn, is created and placed into the urn. This urn will become relevant in the creation of the sampling algorithms implemented in this library.
Another important property is the discrete nature of the random probability measure G. Because of this, we can always write G as a weighted sum of point masses. A useful consequence of this property is its stick-breaking representation, i.e. G can be written as:

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

with $m_k \overset{\text{iid}}{\sim} G_0$ for $k \in \mathbb{N}$ and the random weights constructed as $w_k = v_k \prod_{l<k}(1 - v_l)$ where $v_k \overset{\text{iid}}{\sim} Be(1, M)$.

One can replace the Dirichlet process in (1.1) with any other discrete random probability measure, for example a *Pitman-Yor process*, or PY for short. This is a two-parameter extension of the Dirichlet process, characterized by a discount parameter $\alpha \in [0, 1]$, a strength parameter $\vartheta > -\alpha$, and a base distribution G_0. The Dirichlet process is a particular case of PY when $\alpha = 0$. (For more information, see [4], pp. 855-900.) Nevertheless, we will use the DP as our working example for explanations in the next chapters; theoretical developments are similar if another discrete process is used.

1.3 Dirichlet process mixture model

In many applications in which we are interested in a continuous density estimation, the discreteness can represent a limitation. Oftentimes a Dirichlet process mixture (DPM) model is used, where the DP random measure is the mixing measure for the parameters of a parametric continuous kernel function. Let Θ be a finite-dimensional parameter space and G_0 a probability measure on Θ. The Dirichlet process mixture (DPM) model convolves the densities $f(\cdot|\boldsymbol{\vartheta})$ from a parametric family $\mathcal{F} = \{f(\cdot|\boldsymbol{\vartheta}), \boldsymbol{\vartheta} \in \Theta\}$ using the DP as mixture weights. The

obtained model has the following form:

$$\begin{aligned} y_i|G &\overset{\text{iid}}{\sim} f_G(\cdot) = \int_\Theta f(\cdot|\boldsymbol{\vartheta})\, G(\mathrm{d}\boldsymbol{\vartheta}), \quad i = 1, \dots, n \\ G &\sim DP(MG_0) \end{aligned} \tag{1.4}$$

An equivalent hierarchical model is:

$$\begin{aligned} y_i|\boldsymbol{\vartheta}_i &\overset{\text{ind}}{\sim} f(\cdot|\boldsymbol{\vartheta}_i), \quad i = 1, \dots, n \\ \boldsymbol{\vartheta}_i|G &\overset{\text{iid}}{\sim} G, \quad i = 1, \dots, n \\ G &\sim DP(MG_0) \end{aligned} \tag{1.5}$$

where the *latent variables* $\boldsymbol{\vartheta}_i$ are introduced, one per unit. Since G is discrete, we know that two independent draws $\boldsymbol{\vartheta}_i$ and $\boldsymbol{\vartheta}_j$ from G can be equal with positive probability. In this way the DPM model induces a probability model on clusters of $\boldsymbol{\vartheta}_i$. An object of interest that derives from this model is the partitioning induced by the clustering.
Considering n data points, each $\boldsymbol{\vartheta}_i$ will have one of the k unique values $\boldsymbol{\phi}_j$. An estimation of the number of the unique values is $M \log(n) \ll n$. Defining $\boldsymbol{c} = (c_1, \dots, c_n)$ the *allocation* parameters to the clusters such that $c_i = j$ if $\boldsymbol{\vartheta}_i = \boldsymbol{\phi}_j$, model (1.5) can be thought of as the limit as $K \to +\infty$ of a finite mixture model with K components (recall instead that k is the number of unique values):

$$\begin{aligned} y_i|\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_k, c_i &\overset{\text{ind}}{\sim} f(\cdot|\boldsymbol{\phi}_{c_i}), \quad i = 1, \dots, n \\ c_i|\mathbf{p} &\overset{\text{iid}}{\sim} \sum_{j=1}^{K} p_j \delta_j(\cdot), \quad i = 1, \dots, n \\ \boldsymbol{\phi}_c &\overset{\text{iid}}{\sim} G_0, \quad c = 1, \dots, k \\ \mathbf{p} &\sim \text{Dir}(M/K, \dots, M/K) \end{aligned} \tag{1.6}$$

where $\mathbf{p} = (p_1, \dots, p_K)$ represents the mixing proportions for the clusters and each $\boldsymbol{\vartheta}_i$ is characterized by the latent cluster c_i and the corresponding parameters $\boldsymbol{\phi}_{c_i}$.

1.4 Normal Normal-InverseGamma hierarchy

A very common choice for the DPM model (1.4) is the Normal Normal-InverseGamma (NNIG) hierarchy, opting for a Normal kernel and the conjugate Normal-InverseGamma as base measure G_0. The InverseGamma is the distribution of a random variable Y such that Y^{-1} follows a Gamma distribution[2]. That is, letting $\boldsymbol{\vartheta} = (\mu, \sigma)$, we have:

$$\begin{aligned} f(y|\boldsymbol{\vartheta}) &= N(y|\mu, \sigma^2), \\ G_0(\boldsymbol{\vartheta}|\mu_0, \lambda_0, \alpha_0, \beta_0) &= N\left(\mu|\mu_0, \frac{\sigma^2}{\lambda_0}\right) \times \text{Inv-Gamma}(\sigma^2|\alpha_0, \beta_0). \end{aligned} \tag{1.7}$$

[2]The Gamma distribution $\text{Gamma}(\alpha, \beta)$ has parameters $\alpha, \beta > 0$, and has support over all positive values, which is ideal for variances or their inverse, the precision. Its p.d.f. is

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} \exp\{-\beta x\}$$

with $\Gamma(\alpha)$ being the Euler-Gamma function.

Note that in this model we have a full prior for σ^2 and instead a prior for μ that is conditioned on the value of σ^2. Thanks to conjugacy, the predictive distribution for a new observation $\widetilde{y}$ (which coincides with the marginal distribution for a known datum) can be computed analytically, finding a Student's t (see [5] section 3.5):

$$p(\widetilde{y}|\mu_0, \lambda_0, \alpha_0, \beta_0) = \int_\Theta f(\widetilde{y}|\boldsymbol{\vartheta})\, G_0(\mathrm{d}\boldsymbol{\vartheta}) = t_{\widetilde{\nu}}\left(\widetilde{y}|\widetilde{\mu}, \widetilde{\sigma}^2\right)$$

where the following parameters are set:

$$\widetilde{\nu} = 2\alpha_0, \quad \widetilde{\mu} = \mu_0, \quad \widetilde{\sigma}^2 = \frac{\beta_0(\lambda_0+1)}{\alpha_0\lambda_0}$$

We highlight the definition of the Student's t distribution in the general multivariate case, since it will often come up in calculations. Given the degrees of freedom $\nu > 0$, the k-dimensional location vector $\boldsymbol{\mu}$, and the k-by-k, positive semi-definite scale matrix Σ, the k-dimensional Student's t distribution $t_\nu(\boldsymbol{\mu}, \Sigma)$ has the following p.d.f.:

$$f(\mathbf{x}) = \frac{\Gamma\left(\frac{\nu+k}{2}\right)}{\Gamma(\nu/2)\sqrt{(\nu\pi)^k \det(\Sigma)}} \left(1 + \frac{1}{\nu}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)^{-(\nu+k)/2}.$$

In the univariate case, the scale parameter is written as σ^2. When no location and scale are provided, it is understood that the symbol t_ν refers to the one-dimensional case with standard parameters, that is $\mu = 0$ and $\sigma^2 = 1$.
Going back to the NNIG hierarchy, the posterior distribution is again a Normal-InverseGamma (see [5] section 3.3) thanks to conjugacy:

$$p(\boldsymbol{\vartheta}|y_1, \ldots, y_n, \mu_0, \lambda_0, \alpha_0, \beta_0) = N\left(\mu|\mu_n, \frac{\sigma^2}{\lambda_n}\right) \times \text{Inv-Gamma}(\sigma^2|\alpha_n, \beta_n)$$

with $\bar{y}$ indicating the sample mean of the data and:

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

1.5 Normal Normal-Wishart hierarchy

The multivariate extension of the above hierarchy is the Normal Normal-Wishart (NNW) model:

$$\begin{aligned} f(\mathbf{y}|\boldsymbol{\vartheta}) &= N(\mathbf{y}|\boldsymbol{\mu}, T^{-1}), \\ G_0(\boldsymbol{\vartheta}|\boldsymbol{\mu}_0, \lambda, T_0, \nu) &= N\left(\mu|\boldsymbol{\mu}_0, (\lambda T)^{-1}\right) \times \text{Wish}(T|T_0, \nu). \end{aligned} \tag{1.8}$$

In this case $\boldsymbol{\vartheta} = (\boldsymbol{\mu}, T)$ and the data $\mathbf{y}$ is k-dimensional. Note that the scale is parametrized in a different way compared to univariate case, with the second

unique value being the precision matrix $T = \Sigma^{-1}$, the inverse of the covariance matrix. As such, the Wishart distribution[3] is used instead of its inverse counterpart. Our settings for this hierarchy's hyperparameters are as follows:

- $\boldsymbol{\mu}_0 = \bar{\mathbf{y}}$: the sample mean is often a good starting point for assessing the distribution of the data;
- λ small, e.g. 0.1: being at the denominator of the covariance matrix of the normal, this allows for a large initial variance, which means that the algorithm has more freedom to move where the data dictates right from the start, instead on focusing on a narrow spike which a small variance would produce;
- $\nu = k + 3$ so that the excepted value and median are well-defined;
- $T_0 = \frac{1}{\nu} I_k$, so that the prior expectation of the precision $\mathbb{E}[T] = \nu T_0$ is the identity matrix (same as picking unitary variance in the monodimensional case).

Note that the influence of the hyperparameters on the state values is only direct at the very start of the algorithm, and they will become less and less relevant the more data are taken into consideration. Provided that the algorithm works as intended, it is not imperative to have good parameters in order to get good results.
This hierarchy has similar properties to the NNIG case, namely conjugacy and a marginal distribution with the form of a multivariate Student's t. In particular, the marginal is:

$$p(\widetilde{\mathbf{y}}|\boldsymbol{\mu}_0, \lambda, T_0, \nu) = t_{\widetilde{\nu}}(\widetilde{\mathbf{y}}|\widetilde{\boldsymbol{\mu}}, \widetilde{\Sigma})$$

with:

$$\widetilde{\nu} = 2\nu - k + 1, \qquad \widetilde{\boldsymbol{\mu}} = \boldsymbol{\mu}_0, \qquad \widetilde{\Sigma} = T_0^{-1} \left(\nu - \frac{k-1}{2} \right) \frac{\lambda}{\lambda + 1}$$

while the posterior distribution is:

$$p(\boldsymbol{\vartheta}|\mathbf{y}_1, \ldots, \mathbf{y}_n, \boldsymbol{\mu}_0, \lambda, T_0, \nu) = N\left(\boldsymbol{\mu}|\boldsymbol{\mu}_n, \frac{1}{\lambda_n}\Sigma\right) \times \text{Wish}(\Sigma|T_n, \nu_n)$$

with:

$$\boldsymbol{\mu}_n = \frac{1}{\lambda + n}(\lambda\boldsymbol{\mu}_0 + n\bar{\mathbf{y}}), \qquad \lambda_n = \lambda + n, \qquad \nu_n = \nu + \frac{n}{2},$$
$$T_n = T_0^{-1} + \frac{1}{2}\sum_{i=1}^{n}(\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^T + \frac{\lambda n}{2(\lambda + n)}(\bar{\mathbf{y}} - \boldsymbol{\mu}_0)(\bar{\mathbf{y}} - \boldsymbol{\mu}_0)^T.$$

[3]The Wishart distribution $\text{Wish}(T_0, \nu)$ is a k-dimensional generalization of the Gamma distribution, given T_0 positive semi-definite and $\nu > k - 1$. It has support over all matrices $X \in \mathbb{R}^{k \times k}$ that are positive semi-definite. Its p.d.f. is

$$f(X) = \frac{(\det(X))^{(\nu-k-1)/2} \exp\{-\frac{1}{2}\text{tr}(T_0^{-1}X)\}}{2^{\nu k/2} (\det(X))^{\nu/2} \Gamma_k(\nu/2)}$$

with $\Gamma_k(\alpha)$ being the k-dimensional Gamma function.

Chapter 2

Algorithms

For the task of density estimation, we investigated several Markov chain methods to sample from the posterior distribution of a non-parametric model.
Starting from the hierarchical model (1.4), a first direct approach is simply drawing values for each $\boldsymbol{\vartheta}_i$ from its conditional distribution, given the data and the other $\boldsymbol{\vartheta}_j$. However, as previously discussed, we have high probability for ties among them which can lead to slow convergence, since the $\boldsymbol{\vartheta}_i$ are not updated for more than one observation simultaneously.
For this reason, special attention was paid to the three methods we present in this chapter. These are all iterative *Gibbs sampler* methods, that is, each value is updated by drawing from its own conditional distribution given all other values, repeating these action several times per run. A fixed amount of the initial iterations, known as the *burn-in* phase, will not be counted as part of the state and will be discarded, having the only purpose to allow the method to converge. Moreover, these three methods have a common base structure, sharing the two steps for the sampling of the allocations $\mathbf{c}$ and of the unique values $\boldsymbol{\phi}_c$. The set of allocations and unique values at a given iteration constitutes the *state* of that iteration. As the state is being updated at each iteration, a *chain* is formed and the mean of the state values eventually reaches convergence, as well as the estimate for the data distribution, as we will see in section 3.2. Moreover, all methods can be extended with additional steps for hierarchical extensions. For example, we can place priors to hyperparameters of the centering measure G_0 or to the total mass M.

2.1 Neal's Algorithm 2

In order to speed up convergence in case of ties, Neal first proposed (see [2] section 3 as well as [1] chapter 2) a more efficient Gibbs sampling method based on the discrete model (1.6), but where the mixing proportions $\mathbf{p}$ have been integrated out. We will refer to this method as Neal's Algorithm 2, or `Neal2` for short. Before getting to the algorithm, let us start from the discrete model (1.6). Assuming that the current state of Markov chain is composed of $(c_1, \ldots, c_n)$ and the unique values $\boldsymbol{\phi}_c$ for all $c = 1, \ldots, k$, the Gibbs sampler should first draw a

new value c for each c_i according to the following probabilities:

$$\text{If } c = c_j \text{ for some } j\text{: } \mathbb{P}(c_i = c | \boldsymbol{c}_{-i}, y_i, \boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_k) \propto \frac{n_{-i,c} + M/K}{n - 1 + M} f(y_i | \boldsymbol{\phi}_c) \tag{2.1}$$

where $\boldsymbol{c}_{-i}$ is $\boldsymbol{c}$ minus the i-th component, and $n_{-i,c}$ is the number of c_j equal to c excluding c_i. The transition to the infinite case, that is, to the reference DPM model (1.5), is handled by taking the limit as K goes to infinity in the conditional distribution of c_i, which becomes as follows:

$$\begin{aligned} \text{If } c = c_j \text{ for some } j\text{: } \mathbb{P}(c_i = c | \boldsymbol{c}_{-i}, y_i, \boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_k) &\propto \frac{n_{-i,c}}{n - 1 + M} f(y_i | \boldsymbol{\phi}_c) \\ \mathbb{P}(c_i \neq c_j \text{ for all } j | \boldsymbol{c}_{-i}, y_i, \boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_k) &\propto \frac{M}{n - 1 + M} \int_\Theta f(y_i | \boldsymbol{\vartheta}) \, G_0(\mathrm{d}\boldsymbol{\vartheta}) \end{aligned} \tag{2.2}$$

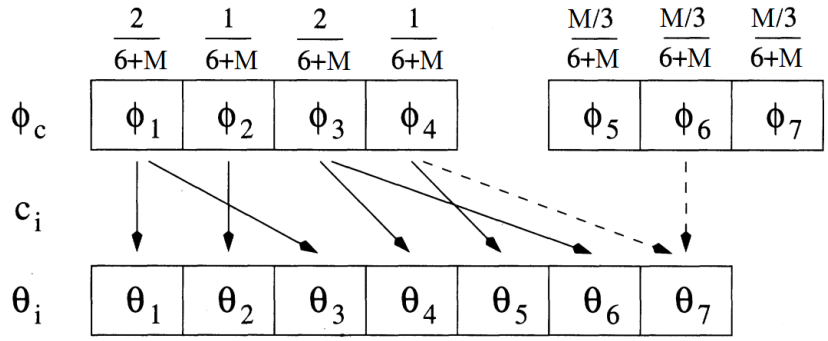
and considering only the $\boldsymbol{\phi}_c$ associated with some observation, keeping the sampling finite and thus computationally feasible. The former expression is proportional to the cardinality of that cluster (excluding the i-th observation), while the latter is instead proportional to the total mass M and represents the probability of creating a new cluster. This is exactly the Polya's urn scheme we touched upon earlier. Moreover, the integral $m(y_i) = \int_\Theta f(y_i | \boldsymbol{\vartheta}) \, G_0(\mathrm{d}\boldsymbol{\vartheta})$ represents the *marginal distribution* of the data points evaluated in y_i.
Let us now introduce the actual `Neal2` algorithm, which works iteratively in two steps, in which we sample $(\boldsymbol{c}_1, \dots, \boldsymbol{c}_n)$ and $(\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_k)$, respectively. First, for each observation i, c_i is updated according to the conditional probabilities (2.2). 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 $\boldsymbol{\phi}_{c_i}$ is created by drawing it from the posterior distribution H_i, given the prior G_0 and the single observation y_i; this means that in this case, a new cluster has been created.
Then, for all clusters, the sampling of their unique value $\boldsymbol{\phi}_c$ is conducted by considering their posterior distribution given the prior G_0 and all observations belonging to that cluster. The probability of setting c_i to a new component involves the computation of the marginal, which is difficult to compute in the non-conjugate case, as well as the sampling from the posterior H_i. For this reason, the algorithm is only used under conjugacy and hence it is possible to exactly compute the integral.

2.2 Neal's Algorithm 8

To handle non-conjugate priors, Neal proposed (see [2] section 6 and [1] chapter 2) a second Markov chain sampling procedure, the `Neal8` algorithm, where the state is extended by the addition of m auxiliary parameters. This technique allows to update the c_i while avoiding the integration with respect to G_0 for the computation of the marginal.
In this case the sampling probabilities for the c_i given all other c_j are:

$$\begin{aligned} \text{If } c = c_j \text{ for some } j\text{: } \mathbb{P}(c_i = c | \boldsymbol{c}_{-i}) &= \frac{n_{-i,c}}{n - 1 + M} \\ \mathbb{P}(c_i \neq c_j \text{ for all } j) &= \frac{M}{n - 1 + M} \end{aligned} \tag{2.3}$$

where the latter probability of creating a new cluster is evenly split among the m auxiliary components, which will also be referred to as the *auxiliary blocks*. Maintaining the same structure as the `Neal2` algorithm, `Neal8` is composed of two steps, where the components of the Markov chain state $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n)$ and $(\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_k)$ are repeatedly sampled.



Graphical representation of the variables: the allocations are visualized as arrows linking each $\boldsymbol{\vartheta}_i$ with either one of the four old clusters or one of the new components (image taken from [2])

The first step scans all the observations and evaluates each c_i. If it is equal to some other c_j, i.e. if the current cluster of observation i is not a singleton, then all auxiliary variables are iid drawn from G_0. If instead the cluster corresponding to c_i is a singleton, then it is linked to one of the auxiliary blocks (i.e. the first one, without loss of generality) while keeping its old unique value $\boldsymbol{\phi}_{c_i}$ (as shown in the above figure), whereas the other blocks are drawn normally from G_0 as before. Then, c_i is updated according to the following conditional probabilities:

$$\mathbb{P}(c_i = c | \boldsymbol{c}_{-i}, y_i, \boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_h) \propto \begin{cases} \dfrac{n_{-i,c}}{n-1+M} f(y_i|\boldsymbol{\phi}_c), & \text{for } 1 \leq c \leq k^- \\ \\ \dfrac{M/m}{n-1+M} f(y_i|\boldsymbol{\phi}_c), & \text{for } k^- + 1 < c \leq h, \end{cases} \tag{2.4}$$

indicating with k^- the number of distinct c_j excluding the current c_i and setting $h = k^- + m$. Again, the probabilities of being placed in an already existing cluster or in a newly created cluster are proportional to the cluster's cardinality (sans observation i) and to the total mass, respectively.
Once all the $\boldsymbol{\phi}_c$ that are no longer associated with any observation are discarded, the algorithm proceeds, for each cluster, with the sampling of $\boldsymbol{\phi}_c$ from the posterior computed with the observations of the specific cluster, similarly to the `Neal2` algorithm.

2.3 Blocked Gibbs

Another Gibbs sampling method that is applicable in the considered DPM model is the one proposed by Ishwaran and James (see [3] section 5), where the prior G is assumed to be a finite dimensional stick-breaking measure, allowing the

update of whole blocks of parameters. A key point of the method is that it does not marginalize over the prior; instead, by grouping more variables together, it samples from their joint distribution conditioned on all other variables. This sampler needs to draw from the following conditionals:

$$\begin{aligned} \boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_k &\sim \mathcal{L}(\cdot | c_1, \ldots, c_n, \boldsymbol{y}) \\ c_1, \ldots, c_n &\sim \mathcal{L}(\cdot | \boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_k, \boldsymbol{p}, \boldsymbol{y}) \\ \boldsymbol{p} &\sim \mathcal{L}(\cdot | c_1, \ldots, c_n) \end{aligned}$$

The drawing of the unique values can also be handled in the non-conjugate case by applying standard Markov chain Monte Carlo methods.
This algorithm is not explored in detail as it has not implemented yet in our library. For a full explanation, see [3].

Chapter 3

Estimates

We recall that the aim of these chain-producing algorithms is to build estimates for both the actual clustering structure of the data and its probability distribution, i.e. their likelihood. We shall now explain how we achieve such estimates. Note that these will be identical regardless of the dimension of the data points.

3.1 Cluster estimation

Suppose we wish to estimate the real clustering of the data, assuming the DPM model holds true. Since the proposed algorithms repeat the same steps for many iterations, they will run through just as many states, i.e. combinations of unique values and clusters, which represent a clustering structure. Therefore, one might think that a first rough estimate for the real clustering could be the *final clustering*, that is, the state values corresponding to the last iteration of the algorithm. However, due to the oscillating behavior of the clusters (as we shall show later on by performing tests), the last clustering is far from being guarateed to be the optimal one. Instead, we chose to implement a *least square* estimate. More specifically, we examine the state values provided by each iteration, then we select the one that minimizes the squared posterior *Binder's loss function*, which gives cost 0 to a pair of points which are correctly assessed to be in the same cluster, and cost 1 to a pair of points which are placed in the same cluster but are actually in different ones. An equivalent approach (see [13] lecture on BNP clustering) 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*, which the sum of squares of the matrix entries. 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 observations) 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 $\bar{D}$:

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

Note that by virtue of the involved matrices being symmetric, the minimization of the latter summation is equivalent to the one computed only for those entries (i, j) such that $i < j$. One can prove that the described algorithms *converge in mean*, but not in the single iterations. That is, the last iterations of the algorithm have no higher probability than the first ones of being the "best" cluster estimate; in fact, the algorithm has quite the *oscillating behaviour* in its single-iteration estimates, starting with the number of clusters at each iteration, as we will see in the last part of the report. Instead, it is the mean of all dissimilarity matrices that converges to the "best" clustering structure: the more iterations are run, the better the approximation provided by the mean becomes. (The number of iterations required is actually surprisingly small: we will show that even a few hundreds are enough to get a good estimate.) Since the mean itself is obviously not a valid dissimilarity, as it is not binary-valued, we choose the one valid iteration matrix that best approximates it. This guarantees the correctness of this least square estimate, at it is the closest available approximation to the mean dissimilarity matrix.

3.2 Density estimation

The other important application of iterative BNP algorithms is the estimation of the density according to which the data points are distributed. This is done slightly differently in both the `Neal2` and `Neal8` algorithms, as the former can exploit the conjugacy of the hierarchical model. Just like for the cluster estimate, the computation will need to access all iterations run by the algorithm. In either algorithm, suppose that iteration k has J clusters, that is, $j = 0 : J-1$. Given a point y, we compute the local estimate of the density, which is built only taking iteration k into account:

$$\hat{f}^{(k)}(y) \;=\; \sum_{j} \frac{n_j^{(k)}}{M+n} f\left(y|\boldsymbol{\phi}_j^{(k)}\right) \;+\; \frac{M}{M+n} m(y) \tag{3.1}$$

where $n_j^{(k)}$ is the cardinality of cluster j. That is, the local estimate is a weighted mean of the likelihood given the unique values $\boldsymbol{\phi}_j^{(k)}$ of cluster j and the marginal distribution $m(y)$ computed in the point. Again, note that the relative 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 the total number of observations, as per usual). The marginal distribution is only known under the conjugacy assumption in the `Neal2` algorithm. In particular, for both an NNIG and an NNW model $m(y)$ is a Student's t as explained in section 1.4 onwards. In the `Neal8` algorithm, $m(y)$ is not available in closed form, and thus it is replaced in the above formula by the following approximation:

$$\hat{m}(y) = \frac{1}{m} \sum_{h=0}^{m-1} f\left(y|\boldsymbol{\phi}_h\right) \tag{3.2}$$

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

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

$$\hat{f}(y) = \frac{1}{K} \sum_{k} \hat{f}^{(k)}(y).$$

Again, this estimate approaches the true posterior density of the data thanks to the convergence in mean of the chain.

Chapter 4

Implementation

The purpose of this library is to provide tools to conduct data analysis through the application of the algorithms described above for the construction of the respective Markov chains and the use of these for density and clustering estimations. A template approach was used, including a set of algorithm classes templatized on the `Hierarchy`, `Hypers`, and `Mixture` classes to allow the use of algorithms with different hierarchies, hyperparameters related to the hierarchy and mixture models.
It currently includes Neal's two algorithms, two Gaussian-based hierarchies (one univariate and one multivariate), and two mixture models.
Unless explicitly stated, the files corresponding to the described objects will be located in the `src` folder and its subfolders.

4.1 Libraries

`bnplib` depends on other libraries such as:

- *Eigen* library, a popular template library for linear algebra, which includes implementations for matrices, vectors, numerical solvers, and related algorithms;
- *Stan Math*, a C++ template library for several math-related tasks; it includes a range of built-in functions for probabilistic modeling, linear algebra, and equation solving. Stan in turn uses the following:
  - the already-mentioned Eigen library (which is the reason why it was chosen over its main alternative, Armadillo);
  - *Boost*, a library that provides support for tasks and structures such as linear algebra, pseudo-random number generation, multithreading, image processing, regular expressions, and unit testing;
  - Intel's *Threading Building Blocks* (TBB) library, to write parallel C++ programs that take full advantage of multicore performance;
  - *Sundials* library, a suite of nonlinear and differential or algebraic equation solvers.

- *Protocol Buffer* library, or Protobuf for short, which was developed by Google and provides a fast serialization mechanism for structured data and extensible code generators for different programming languages.

These libraries exist in the `lib` subfolder.
We also implemented a Python interface for the library, the details of which will be discussed in section 4.4). The following libraries were needed:

- *Protobuf*, to interface data between C++ and Python;
- *pybind11*, a header-only library that exposes C++ types in Python and vice versa;
- *NumPy*, a popular library which provides high-performance multidimensional array objects, and tools for working with them;
- *SciPy*, another widespread library containing modules for numerical integration, interpolation, optimization, linear algebra, and statistics;
- *Scikit-learn*, a free machine learning library built on top of SciPy;
- *Matplotlib*, for creating plots.

First of all, we shall describe the auxiliary classes that are used as template arguments for the algorithms. Probability distributions and random sampling are handled through the `Stan` library, while `Eigen` was exploited for the creation of the necessary matrix-like objects and the use of matrix-algebraic operations throughout the code.
Some main test files are present in the root folder of the library: `maintest_uni.cpp`, `maintest_multi.cpp`, and `maintest_nnws.cpp`, the latter of which we used for several time comparisons in the multivariate NNW hierarchy. In the `README.md` of the project one may find some basic examples of usage of these files.

4.2 Model classes

The *mixture classes* (alias in the algorithm: `Mixture`) contain all information about the mixing part of the BNP algorithm, namely the way of weighing the insertion of data in old clusters with respect to the creation of new clusters. They are implemented starting from the abstract class `BaseMixture` which provides the common interface for those objects. The class has methods that provide probability masses for the two aforementioned events: `mass_existing_cluster()` and `mass_new_cluster()`. We implemented two derived classes from `BaseMixture`: the `DirichletMixture` and the `PitYorMixture`. The derived classes have their own parameters (including their respective getters and setters) and could be extended by adding prior distributions on those, for instance by creating further classes that inherit from them.

The *hyperparameters classes* (alias: `Hypers`) contain information about the hyperparameters of the hierarchy, including their values (if fixed) or their prior distributions (if not). Unlike all other types of classes, these are not based on inheritance from a common abstract base class, since their internal structure is vastly different. Moreover, each of them is to be used in conjunction with

a single hierarchy class anyway, so there is less need for a inheritance-based implementation. We implemented the `HypersFixedNNIG` class, which contains fixed hyperparameters for an NNIG hierarchy, and the `HypersFixedNNW` class for an NNW hierarchy. Both classes provide setters and getters for parameters with validity checks for the inserted values.

Finally, the *hierarchy classes* (alias: `Hierarchy`) are implemented starting from the abstract template class `HierarchyBase<Hypers>`. They represent the form of the hierarchical model in a generic Gibbs sampler BNP algorithm, namely the likelihood of data points, including their parameters, the unique values. Since only data points that are in the same cluster as each other share unique values, this means that for each cluster there will be a hierarchy object, that is, a single set of unique values with their own prior distribution attached to it. These values are part of the Markov chain's state chain which develops as the iterations of the algorithm increase, updating them providing the data related to the specific hierarchy and their prior distribution. These are simply referred to as the *hierarchy state*; they are not the same thing as the *state of the chain*, which is composed of all allocations and unique values across all hierarchy objects. A hierarchy class stores the current unique values for the cluster it represents in the member `state`, a vector of parameter matrices. Since the prior distribution for the state is often the same across multiple different hierarchies, the hyperparameters object is accessed via a shared pointer, and this is why `Hypers` is required as a template parameter for the class.
The constructor of a hierarchy class contains the shared pointer to create the hyperparameters object with, and initializes the hierarchy state to default values. These classes also contains methods to:

- evaluate the marginal distribution (in the hierarchies where it is known in closed form) and the likelihood in a given set of points, conditioned on the current `state` values: `eval_marg()` and `like()`;
- compute the posterior parameters with respect to a given set of observations: `sample_given_data()`;
- generate new values for the `state` both according to its prior (`draw()`) and to its posterior (`normal_gamma_update()` or similar) centering distribution;
- getter and setter class members, as with the other classes. The peculiarity of the setter for `state` is that it also accept a boolean flag `check`. If the setter is called with the flag set to `true` (which is also its default value), it mandates the calling of the `check_state_validity()` utility, which is specific for each hierarchy and raises an error if the state values are not valid with respect to their own domain (for instance, a negative value for the variance). This type of check is only done in `set_state()`, and not in the functions that generate new values for the state, since these functions always generate coherent values, which makes any check redundant. This saves some computational time, especially in the multivariate case where an expensive check on the precision matrix must be performed to assess its positive semi-definiteness. For the same reasons, the setter is called with the flag set to `false` also in the algorithm functions (see below).

The derived hierarchy classes are `HierarchyNNIG`, which represents the Normal Normal-InverseGamma hierarchy for univariate data, and `HierarchyNNW`, which represents the Normal Normal-Wishart hierarchy for multivariate data. As mentioned in section 1.4 onwards, the `state` in `HierarchyNNIG` holds the values for $\boldsymbol{\phi} = (\mu, \sigma)$, i.e. location and scale, while in `HierarchyNNW` one has $\boldsymbol{\phi} = (\boldsymbol{\mu}, T)$, i.e. location and precision parameters. Recall that both hierarchies are conjugate, thus the Neal2 algorithm may be used with them.

4.3 Algorithm classes

The algorithms studied and discussed in the theoretical section all share the same structure, so we decided to build an abstract class for a generic Gibbs sampling iterative BNP algorithm, the `Algorithm` class:

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

All algorithms of this form can be built as derived classes from this one. It is constructed by the following function:

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

As mentioned, the first argument initializes the passed `Hypers` object through a shared pointer. `init` is assigned to the `init_num_clusters` member, which states the number of clusters in the initialization phase of the algorithm. If a value is not provided, it will be set equal to the data size in the constructor body, therefore placing one object in each cluster. Dimensionality checks also take place, in order to verify that all passed objects have coherent dimensions with each other.
The `Algorithm` class contains the integer class members `maxiter`, the cumulative number of algorithm iterations, and `burnin`, the number of initial iterations in the burn-in phase, which will be discarded. These integer values, including the seed for the `rng` object (of type `std::mt19937`, a basic Mersenne Twister random number generator provided by the standard C++ library) and the `n_aux` parameter that indicates the number of auxiliary blocks in `Neal8`, are not present in the class constructor since they are usually left with their default values, though they can still be changed through their respective setters. If no other values are provided, `maxiter` is initialized to 1000, `burnin` to 100, and `n_aux` to 3, values that we have assessed as sufficient for a good approximation after performing several tests. Note that changing these values after running the algorithm (see below for the `run()` function) has no effect as far as the execution of the algorithm is concerned.
Moreover, the class contains several data and values 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;
```

The matrix of row-vectorial data points is given as input to the class constructor by the user. Data points were chosen to be matrix rows, even though points in a space are usually represented as column vectors, because this is the way data are usually stored in comma- or space-separated values files. In fact, the `utils.hpp` file contains the `read_eigen_matrix()` utility, which returns an `Eigen::MatrixXd` after reading values from a given filename. This works for both data matrices and grid matrices on which evaluate the estimated density (see section 4.5 for the implementation details).
The algorithm will keep track of the labels representing assignments to clusters via the `allocations` vector. For instance, if at any point one has `allocations[5] = 2`, it means that datum number 5 is associated to cluster number 2, with indexing starting at zero. Similarly, we store the `cardinalities` of the current clusters and `unique_values`, which is a vector of Hierarchy objects which identify the clusters and in which the corresponding unique values are stored in the `state`, as mentioned in the previous section. The three aforementioned vectors are initialized with null values and empty `Hierarchy` objects, and will be filled with proper values while the algorithm is running.
The `run()` method is the same from all derived classes, given that all implemented algorithms share the same general structure:

```
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();
}
```

That is, a Gibbs sampling iterative BNP algorithm generates a Markov chain on the clustering of the provided data running multiple iterations of the same `step()`. The latter is further split into substeps, each of which updates specific values of the state of the Markov chain, which we recall to be composed of the allocations vector and the unique values vector. Substeps are then overridden in the specific derived classes. In particular, among the studied algorithms, only the currently unimplemented blocked Gibbs algorithm exploits the `sample_weights()` function. Moreover, `update_hypers()` only has an effect when the hyperparameters are not fixed. Therefore this function is not currently used in the library either, since we only have `Hypers` classes representing

fixed ones at the moment. The collector argument of `run()` and the functions `start()`, `save_state()`, and `finish()` deal with storage of the chain state, which we will discuss later in the appropriate section 4.4.
We shall now describe the features of the two implemented derived classes: `Neal2` and `Neal8`.

4.3.1 `Neal2`

As discussed in section 2.1, the `Neal2` algorithm exploits conjugacy, thus the class requires specifically implemented hierarchies, in which the marginal distribution of the data with respect to $\boldsymbol{\vartheta}$ is available in closed form. This class implements the substeps of the `run()` function as follows:

- In `initialize()`, a number of clusters equal to the provided initial value are created, and data are randomly assigned to them, while making sure that each cluster contains at least one. Assignment to a cluster means that the `allocations` entry for the datum is set equal to the number of the cluster, as explained earlier.
- In `sample_allocations()`, a loop is performed over all observations $i = 1 : n$. The vector `cardinalities` is filled at first, with `cardinalities[j]` being the cardinality of cluster j. Then, the algorithm mandates that `datum = data.row(i)` be moved to another cluster. A vector `probas` is filled with the probabilities of each cluster being chosen, including a new one, as shown in (2.2): computations involve the `cardinalities` vector, the mass probabilities defined by the `Mixture`, the likelihood `like()` evaluated in `datum` to compute the probability of being assigned to an already existing cluster, and the marginal `eval_marg()` to compute the probability of being assigned to a newly generated cluster. Then, the new value for `allocations[i]` is randomly drawn according to the computed `probas`. Finally, four different cases of updating `unique_values` and `cardinalities` are handled separately, depending on whether the old cluster was a singleton or not and whether or not `datum` is assigned to a new cluster. Indeed, in such a case, a new $\boldsymbol{\phi}$ 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). 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 $\boldsymbol{\phi}$ 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.

4.3.2 `Neal8`

The `Neal8` algorithm is a generalization of `Neal2` which works for any hierarchical model, even non-conjugate ones, by adding adjustments in the allocation

sampling phase to circumvent non-conjugacy. However, the unique values sampling phase remains unchanged. For this reason `Neal8` is built as a derived class from `Neal2`. In addition to the members already defined in the parent classes, the following are added:

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

These are related to the auxiliary blocks, which are unique to this algorithm. A setter exists for `n_aux` which also updates the `aux_unique_values` vector to the correct number of blocks.
We now proceed to describe the new implementation of `sample_allocations()` in this class. A loop is first performed over all observations $i = 1 : n$ and `cardinalities` is filled, just like in `Neal2`. Now, 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, new ϕ values, for `datum` needs to be drawn. The vector `probas` with `n_clust+n_aux` components is filled with the probabilities of each ϕ being extracted, computed as shown in (2.4); this formula also involves the auxiliary blocks. The new value for `allocations[i]` is then randomly chosen according to the computed `probas`. Finally, four different cases of updating `unique_values` and `cardinalities` 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 `datum`.

4.4 Collectors

After an algorithm is run, one may be interested in an estimate of the density given a grid of points, and/or in an identification of a partition through the clustering obtained in the algorithm, which we sometimes call "best clustering", in the sense we described in section 3. In any case we need to be able to save and to retrieve the information of each iteration of the algorithm, such as allocations and unique values, which characterize the state of the chain. This values must therefore be stored in an appropriate data structure, preferably one which is independent of `Algorithm`, so that we do not have to save the entire chain of `State` objects as a member. For this reason, we have implemented the collector classes. A *collector* is an external class meant to store the state of the Markov chain at all iterations as a list of `State` objects. Their implementation is based on the Protobuf (Protocol Buffers) library, which allows automatic generation of data-storing C++ classes by defining a class skeleton in the `chain_state.proto` file located in the root folder. This also allows easy interfacing with other programming languages such as R and Python.
We built the template for our collector classes as follows:

```
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;
}
```

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. The corresponding C++ and Python classes are automagically generated off of this skeleton via the `protoc` compiler.
We have implemented two types of collectors: `FileCollector` and `MemoryCollector`, both of which are derived classes from the `BaseCollector` abstract class. The former saves the `State` objects into appropriate binary files, while the latter places them in memory into a deque. In particular, the `MemoryCollector` does not "hard write" chain states anywhere and all information contained in it is destroyed when the main that created it is terminated. It is therefore useful in situation in which writing to a file is not needed, for instance in a main program that both runs the algorithm and computes the estimates. Instead, the contents of a `Filecollector` are permanent, because every state collected by it remains ever after the termination of the main that created it, in Protobuf form, in the corresponding file. This approach is mandatory, for instance, if different main programs are used to run the algorithm and the estimates, for instance in the Python interface (more on this later, in section 5). Both collectors store the current size of the chain, i.e. the number of the stored `State` objects, which is constantly updated during the run after each performed iteration, and `curr_iter`, an integer that acts as a cursor, useful when reading the chain state-by-state.

4.4.1 Writing and reading

In the main program, a collector whose type is chosen at runtime is instantiated before running the algorithm, and a `BaseCollector` pointer points to the collector object, exploiting polymorphism. Then, the `BaseCollector` pointer is passed to the `run()` method. Let us take a closer look at the *writing* procedure in it. In the specific case of `FileCollector`, we first pass a string to the constructor, which initializes the `filename` where the chain will be saved. Then, in `run()`, the `start()` method creates a so-called open file description that refers to the file, as well as a stream object that writes to it via a Unix file descriptor. After each iteration, the `collect()` function is called which takes as input the current state in Protobuf object format and writes it into the file. At the end of `run()`, after all iterations have been performed, `finish()` closes the file descriptor and the corresponding file. The writing procedure is similar when using a `MemoryCollector`, except that `collect()` inserts the current iteration's state in Protobuf-object form into the deque.

Once the algorithm's run is completed, one may proceed with the density and clustering estimates. Both of these require *reading* from the collector, since they require knowledge of the entire chain of states. Therefore, the aforementioned `BaseCollector` pointer is also passed to the estimate functions, inside of which the method `get_chain()` is called, which returns the whole chain in deque form. In the case of `MemoryCollector` this simply means returning the currently stored `chain` protected member, while in the case of `FileCollector` the chain is read state-by-state from the binary file, converted back into Protobuf format and returned as a deque.
In fact, two alternatives are possible for reading: the above method of using `get_chain()`, and reading one state at a time directly in the estimates functions when necessary, without having to save back the whole chain. This is achieved using the function `get_next_state()`, which returns one state at a time based on the current position given by the cursor `curr_iter`, which is increased by 1 with each call. With such an implementation, in both estimate functions' loops over the iterations, `get_next_state()` is called at each round, which in turn calls the `get_next()` protected method, specifically implemented for each collector, and the state relative to the current round is returned. This is easily achieved when using a `MemoryCollector`, but in the case of `FileCollector`, the `State` object must be read from the file itself. However, unlike in the cases using `MemoryCollector`, if the estimate functions are executed in a different source file than the one where the algorithm is run, a `FileCollector` must be used, but in the second source file the information on the size of the chain and of the objects is unavailable, since the actual C++ object was destroyed when closing the first one. This information is crucial in order to correctly read from the produced binary file. Therefore, our library currently uses the `get_chain()` method which re-creates the whole chain even from a `FileCollector` instead of the state-by-state method. In order to use the latter, one would have to store additional information about object sizes in the file itself.
In addition, in the cluster estimate utility, even though all dissimilarity matrices can be computed via state-by-state reading, one needs random access to retrieve the `State` object corresponding to the best clustering. In the case of `FileCollector`, since the current implementation re-creates the `chain` deque, one can simply access `chain[i]` where `i` is the iteration that minimizes the squared error. If the above reading mode were to be implemented, one would need to use a getter `get_state(unsigned int i)`, which would make necessary to re-read the file up to this iteration with multiple calls to `get_state()`. In this case, optimization may be achieved by saving the byte size of the individual Protobuf objects, so that one can recover a specific state directly by jumping at the correct location in the file without having to read and return all previous iterations.

4.5 Estimate functions

The cluster and density estimation described in chapter 3 are implemented in the following methods in the `Algorithm` base class:

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

Both functions exploit the chain saved in the passed collector object. The former loops over all `State` objects in the chain to compute and save the dissimilarity matrix for each of them, while incrementally updating the average dissimilarity matrix. As previously noted, the matrix need not be filled in its upper triangular part, which can effectively be ignored, thus saving over half of the computation time.
Storing the matrices of all iterations is mandatory, since the function needs to pick the one which is closest, in terms of Frobenius norm, to the yet-unknown average dissimilarity. Since these matrices are quite large in dimension (n-by-n) and hundreds of them need to be stored, we used the `Eigen::SparseMatrix` class instead of its usual `Dense` counterpart. After that, the error for each matrix (or rather for its lower triangular part) is computed, and the function returns the index of the matrix which has the least error after saving the corresponding `State` object in the `best_clust` class member.

The density estimation function is also implemented in the base `Algorithm` class, despite `Neal2` and `Neal8` having different formulas for it, because both estimates are actually identical except for the last addendum involving the marginal distribution, insofar that they are the average over all iterations of local (i.e. iteration-specific) density estimates. They therefore share a large part of the code, namely the loop over all states of the chain, in which the cardinalities vector of the current state is recomputed and a `Hierarchy` object to compute the likelihood with is rebuilt. The likelihood will then be multiplied by the mixture-dependent weights, which in turn depend on the computed cardinalities. The only difference between both algorithms is enclosed in the `density_marginal_component()` subroutine, which has a specific implementation for each of them: `Neal2` uses the closed-form marginal by exploiting the conjugacy of the model, while `Neal8` has to compute an arithmetic mean on a small random sample. The other parameter of `eval_density()` is a `grid` of Eigen points in which the density will be evaluated. After `eval_density()` is run, this grid is stored together with the evaluated points themselves in the `density` member object, which is an `std::pair` of Eigen matrices.

We also implemented two writing utilities, which save data from the class into text files in order to ease exportation to other programs, machines, or even the Python interface itself:

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

They can be called as need be from any main file. Before writing to a file, each of these functions checks an appropriate boolean flag, which are called respectively `clustering_was_computed` and `density_was_computed`, which is changed to `true` only at the end of the corresponding estimation function; if such flag is still `false`, an error message is printed. Otherwise, they write the information stored in the appropriate member classes to a file in a comma-separated value format.

4.6 Algorithm factory

In order to have the possibility of a runtime choice of the algorithm, we implemented the `Factory` class. An *object factory* allows to choose at runtime one of several object types for a given variable, provided that these objects derive from a common abstract base class, whose name is passed as a template parameter with the alias `AbstractProduct`. An object factory is usually implemented as a singleton and stores a list of *builders*, i.e. functions that each create a different kind of object, into the private `storage` class member. Each of these builders can be as simple as a function returning a smart pointer to a new instance, and has an identifier (e.g. a string) corresponding to the specific object they create: `storage` is therefore an Identifier-Builder map. In order to have the choice of creating one of multiple objects, `storage` must first be filled with their corresponding builders; this can be done in a main file or in an appropriate separate utility function. Since the constructors of the implemented algorithm classes may take different numbers of parameters as input, we chose to templatize the factory with a variadic template in addition to the `AbstractProduct` type. This allows passing any number of parameters of any type to the constructors of the objects.
Theoretically, it would be possible to use the same `Factory` implementation file to create multiple factories which each produce different categories of objects, since templates specialized with different arguments (`AbstractProduct`) are treated as different objects altogether. However the abstract product, which in our case is one of two algorithms, is defined with specific hierarchy, mixture, and hyperparameters classes, all of which must be known at compile time. Therefore one could not have different independent factories that each generate a piece of the model. In order to choose every class at runtime, it would be required to add all possible combinations of algorithms, hierarchies, and mixtures to the `storage`, so that all possible `AbstractProduct`s are known at compile time. This is possible with the current number of available objects, but as the library expands with new classes being added, this method would not scale well. Therefore, at the moment only the choice of the algorithm type is available at runtime.
Another difficulty for full runtime choices is that the hyperparameters classes do not currently have a base class, since their implementations heavily depend on the hierarchy they support and therefore have little to no common structure.

Chapter 5

Python interface

In this section we will talk about the Python interface we implemented for the library, for a more user-friendly and versatile use of the code. We first note that thanks to the algorithm factory described in the previous section, the user can choose the desired algorithm directly from their Python script. The files for this interface are located in the `src/python` folder.
This interface is made possible by two main pieces of software: the `pybind11` Python package, and again the Protocol Buffers library. `pybind11` allows transporting of C++ objects into a C++ library that can be read as if it were a Python library. In particular, the `cpp_exports` subfolder contains several independent source files each containing a function that fulfills a specific role. This allows the user to be able to run the algorithm and execute the estimate at different points in time, since these two actions are completely independent – this is actually the main reason for why a `FileCollector` was implemented in the first place. Such independence is possible because after using the run unit, the Markov chain is automatically saved to a `FileCollector`, which can then be read and deserialized thanks to the Python interface of the Protobuf library. More specifically, the `chain_state.proto` file that was used to generate the `State` class in C++ is also used to generate the same exact class in a Python file, again by running the Protobuf compiler `protoc`; the created file is called `chain_state_pb2.py`. All these units are included into the `exports.cpp` file, in which the macro `PYBIND_MODULE` is invoked to create the Python version of the library by passing the created function objects by reference:

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

(Note that the units included in the library must have a fixed hierarchy and mixture, since the choice at runtime for these objects is not available yet, as previously noted.) The library is then compiled into a shared `.so` C++ library by calling the `Makefile` rule, `pybind_generate`. After the library is created, one can simply `import bnplibpy` in any Python script after adding its file path to the `PYTHONPATH` environment variable.

5.1 Using the interface

In particular, a Python interface file, `bnp_interface.py` was created to automatically import the library and implement several useful tools:

- `get_multidim_grid()` creates an hypercube grid of arbitrary side, dimension and step, which is useful for evaluating a higher-dimension density estimate;
- `deserialize()` exploits the aforementioned common interface provided by Protobuf to read a Markov chain from a `FileCollector` given its name and turn it into a list of `State` objects;
- `chain_barplot()` loops over the Markov chain unpacked by `deserialize()` and produces a barplot which shows the distribution of the number of clusters over all saved iterations of the chain;
- `plot_density_points()` and `plot_density_contour()` both take a density evaluation file as input and plots them if possible, i.e. if the given density has the correct dimensions: 1D and 2D for the former, which is a regular function graph, and 2D for the latter, which is a map of the estimated contour lines of the function;
- `plot_clust_cards()` takes a clustering `.csv` file as input, most likely the best clustering computed and stored via the `cluster_estimate()` method of the `Algorithm` class, and plots the cardinalities of clusters inside it;
- `clust_rand_score()` computes the so-called *adjusted Rand index* (ARI), or score, between a given predicted clustering and true clustering, which is a value in $[0, 1]$ that represents a measure of similarity between both clusterings. In particular, it is the proportion RI of correct decisions made by the clustering algorithm with respect to the true clustering: the higher the score, the better the estimation. Such proprortion is then adjusted for chance in the following way: $ARI = (RI - \mathbb{E}[RI])/(\max(RI) - \mathbb{E}[RI])$. (See [6] for further details.)

The user can then call each of these tools and the ones in `bnplibpy` at will in their scripts. For instance, one may want to run the algorithm and get the Markov chain, visualize the distribution of clusters via `chain_barplot()`, then compute the estimates; or maybe do all 3 at the same time. A typical Python script that uses this library looks as follows (extracted from `console.py`):

```
from bnp_interface import *

# Initialize parameters
mu0 = 5.0
lambda_ = 0.1
...

# Write file names
datafile = "csv/data_uni.csv"
collfile = "collector.recordio"
...
```

```
# Run algorithms, estimates, and plots
bnplibpy.run_NNIG_Dir(mu0, lambda_, alpha0, beta0, totalmass, datafile, algo,
    collfile, init, rng, maxit, burn, n_aux)
chain_barplot(collfile, imgfilechain)
bnplibpy.estimates_NNIG_Dir(mu0, lambda_, alpha0, beta0, totalmass, grid, algo,
collfile, densfile, clustfile, only)
plot_clust_cards(clustfile, imgfileclust)
```

Chapter 6

Performance and optimization

In this chapter, we will justify some choices that we made in the code that increase efficiency. In the process, we will show and analyze the output of some optimization tools we used in order to test the performance of this library, as well as some of the aforementioned choices.

6.1 Compiler flags

When calling the `g++` complier from the `Makefile`, we used the following optimization flags, which instruct the compiler to maximize the code efficiency:

```
-march=native -O3 -msse2 -funroll-loops -ftree-vectorize
```

In order to test their effects, we ran the `maintest_nnws.cpp` test file (which has the standard combination of the `HierarchyNNW<HypersFixedNNW>` plus the `DirichletMixture` class) using $n = 10$ 5-dimensional data points and a 7-point grid for the density evaluation, with and without the above flags, while timing both the algorithm execution and the density estimate with the help of the `chrono` library. We use $n = 10$ because in such a high dimension, even computation times on a handful of points become extremely large and can lead to overflow. The obtained average times in shakes (a metric unit of time equal to 10 nanoseconds) were as follows:

	`run()`	`eval_density()`	total
without flags	521762512	182770170	704532682
with flags	39615848	10353269	49969117
speedup	13x	17x	14x

As one can see, the speedup resulting from the addition of these flags is hugely noticeable.

6.2 In the `Algorithm` class

One might note that the cluster cardinalities need not be part of the state, since they can be computed at any time from the allocations. Despite that, we

store the `cardinalities` of the current iteration in a vector, which is first filled when the initial clusters are being built in the `initialize()` function, then updated after the repositioning of each datum in `sample_allocations()`. This is because their explicit computation is needed in `sample_allocations()` in order to be able to compute the masses of the current clusters. The alternative is rebuilding the vector at each call of `sample_allocations()`, which is $O(n)$; this is the same cost as the worst-case scenario of storing them externally, which can potentially happen in case 2. In particular, if the first cluster is a singleton and the current datum moves from it to any other cluster, the whole vector must be shifted to the left by 1. Therefore, the external storage is clearly a better choice.

A second point involves the usage of sparse Eigen matrices instead of dense ones in `cluster_estimate()`. It is mandatory to store the dissimilarity matrix for each of the K iterations must be stored, so that the Frobenius norm of its difference with the mean dissimilarity can be computed. These matrices add up to a total of Kn^2 entries. When the data size starts being moderately large, the memory usage becomes huge if using regular dense matrices. On the contrary, the `Eigen::SparseMatrix` class stores data points as a double vector of position and value, therefore greatly reducing the number of entries in the object, from n^2 to $2n$. The dissimilarity matrices are indeed sparse objects, since their upper triangular part is never filled, and several of the entries in the lower triangular part may be zero as well. In order to fill this type of matrix, a vector `triplets_list` of `Eigen::Triplet` objects must be used which stores each entry's position and value; the `setFromTriplets()` method of the matrix is then used by passing `triplets_list` as argument. In order not to waste time reallocating massive amounts of data, a `triplets_list.reserve()` call, and an estimate for the number of entries, are thus desirable. Suppose that the n data points are evenly distributed into k clusters. In such a case, the number of nonzero entries for the corresponding dissimilarity matrix (without the upper triangular part) is

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

whilst the number for unbalanced clusters is slightly higher. Therefore, $\frac{n^2}{4}$ is a good estimate for the number of entries. This number is very close to the true amount if the number of clusters is small (e.g. $k = 2, 3, 4$), which it often is. (Note that one could theoretically compute the exact number of nonzero entries of the dissimilarity matrix, but this would require additional computational costs which is not warranted for the size of a single vector.) This relatively large amount of reserved space does not significantly impact the overall memory usage, since the `triplets_list` vector is destroyed after each dissimilarity matrix is computed.
The usage of sparse matrices instead of dense ones does not hinder computational time either, since the Eigen library also heavily optimizes computations with sparse objects. Nevertheless, an efficiency test was performed, this time using the Python interface, on the already implemented Python tests 1-6 (see section for full details) with 500 algorithm iterations and 100 burn-in ones, using sparse matrices first and then the same operations but with dense matrices. The results were as follows, again in shakes:

	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

As one can see, the performance is on average nearly identical for both types of implementation, with a difference of only less than 10% for the multivariate tests 5 and 6 in favour of the dense case.

Finally, one more possible improvement for the current implementation is using a completely different approach of storing the data, e.g. with an `std::map` in which each entry is a cluster that holds objects representing the data points themselves, instead of just storing the allocations labels. This may be a way to erase and add clusters more efficiently. Testing this assumption would be very difficult, since it would require implementing all data structures from scratch; nonetheless we think that using vectorial structures is more efficient for everything else, and therefore the correct choice.

6.3 In the `Hierarchy` classes

It was mentioned earlier that our custom implementation of the data likelihood in `HierarchyNNW` was more efficient than the one provided by the Stan library. The difference becomes more and more relevant as the data dimension increases, since there are matricial operations involved. We ran some tests, again in `maintest_nnws.cpp`, to quantify the time saved, using the same 5-dimensional dataset and grid as before. Note that the `like()` function is used not only in density estimation, but also in the allocation sampling phase of the algorithm, thus it affects `run()` as well. We include the average results:

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

6.4 Profiling analysis

`callgrind` is a command-line profiling tool that observes the running of the executable it is given as argument and records the call history among all functions that were called in it. The output is then saved in a `callgrind.out` file, which can be read with the use of the KCacheGrind GUI. The following two commands were run, each on the C++ main test file for both the univariate case (with `HierarchyNNIG<HypersFixedNNIG>`) and the multivariate case (with `HierarchyNNW<HypersFixedNNW>`):

```
valgrind --tool=callgrind ./maintest_uni csv/data_uni.csv neal2 memory
valgrind --tool=callgrind ./maintest_multi csv/data_multi.csv neal2 memory
```

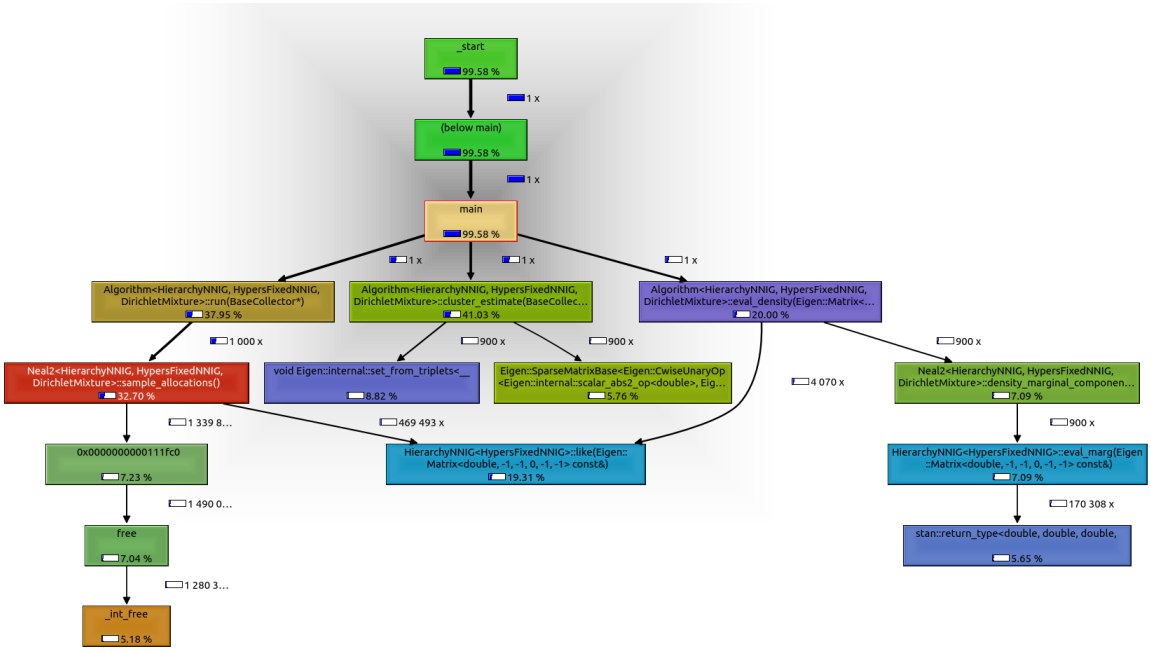


Figure 6.1: `callgrind` dependence graph for `maintest_uni.cpp` (univariate NNIG case).

The output of the univariate case produced the above dependence graph (figure 6.1). The percentage inside the square is the proportion of the total time spent in that particular function across all calls to it. The number followed by x tells instead the number of times that function was called.
We are first interested in the partitioning of the computation time between the different methods. We note that `run()` has slightly less than 40% of the total time, while `eval_density()` has around 20%, and `cluster_estimate()` slightly more than 40%. As for the lower-level operations in `run()`, almost all the time is spent in `step()`, more specifically in `sample_allocations()` (33%). This also means that the cost of collecting the states into the `MemoryCollector` is irrelevant. Moreover, `like()` is the most significant low-level utility function in the library, taking a total of 20% of the total time across both `run()` and `eval_density()`.

Let us now compare these results with the multivariate case, as shown in figure 6.2 in the next page. As for the general composition, `run()` has now become less than 10% of the total, with `sample_allocations()` taking roughly half of that amount, while the `eval_density()` part reaches almost 90%. This is to be expected since nearly all of the time is spent likelihood and marginal computation, which is much harder in 2D than in the univariate case due to the introduction of matrix algebra. We can also note that as a side effect of the growth of the density estimation part, the percentage of time spent in `cluster_estimate()` is now irrelevant, which makes sense since the computational costs of building a dissimilarity matrix is almost identical in all dimensions. The most costly part

of the density estimation is the unoptimized `eval_marg()` (more than 60%), presumably due to the presence of matrix inversions and Student's t evaluations, both of which require expensive Cholesky decompositions. On the contrary, we

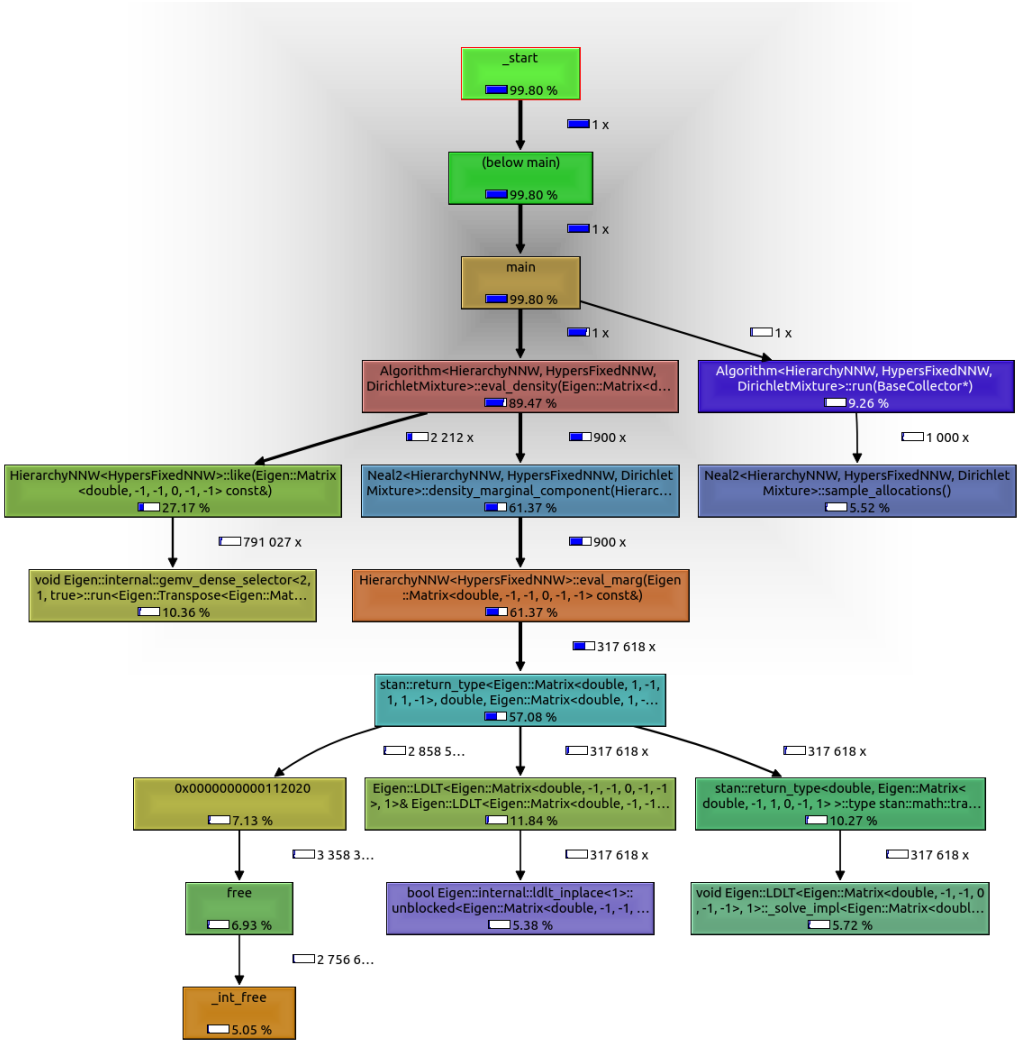


Figure 6.2: `callgrind` dependence graph for `maintest_multi.cpp` (multivariate NNW case).

note that the usage of `like()` is in the same order of magnitude as before (27%). Since the estimate algorithm is the same as before, this means that `like()` is optimized enough for it to withstand the jump to the multidimensional case better than the other functions.
As for a final comparison with the case lacking the compiler flags described in section (6.1), the optimization mainly impacts the multivariate case, with `run()` being the main recipient, going from a whopping 58% to the already mentioned 10%. In the univariate case, everything evenly benefits from the speedup, with

the only relevant difference being for `like()` which goes from over 30% to 20%.

6.5 Limits

Despite the optimizations in the `cluster_estimate()` function, the storing of a number of large matrices still requires a considerable amount of memory. In general, the Protobuf structures also become overwhelmingly large if the data dimension increases. For these reasons, we were not able to run the multivariate Python tests (see chapter 7.2) in higher dimensions such as $d = 10$ and $d = 20$ on our machines.
Another problem related to large dimensions arises when trying to run the `Neal8` algorithm with 5-dimensional data. At one point, the computed covariance matrix in one of the functions of the hierarchy becomes not symmetric. Since a covariance/precision matrix must be symmetric and the Stan library implements a check for this property when passing such matrices as arguments to the density functions, the execution stops. This error is likely due to the fact that the more the dimension of a space increases, the more its points will be far apart from each other – this is the so-called "curse of dimensionality". Since variance is a form of distance, the huge gap between points leads to a covariance matrix with large entries; due to the increased order of magnitude of these numbers, the floating-point approximations which are normally too small to appear become significant. The hierarchy state is actually a precision matrix, which is the inverse of covariance and therefore should have very small values, but the computation of some covariance matrices is still required at some points.

Chapter 7

Results

7.1 Sensitivity analysis

The following tests were performed to observe the dependence of the univariate NNIG model on its several parameters. This analysis was conducted on $n = 100$ observations, half iid sampled from a $\mathcal{N}(4, 1)$ and half from a $\mathcal{N}(7, 1)$. We chose the prior parameters for the model as follows: $\mu_0 = 5, \lambda_0 = 1, \alpha_0 = 2, \beta_0 = 2$. The `Neal8` algorithm with $m = 3$ auxiliary blocks was run for 20000 iterations, and the first 5000 were discarded as burn-in, for a total of $K = 15000$ valid iterations. We will keep these parameters values fixed unless explicitly stated. The following test data were all saved to `.csv` files and used for the realization of plots with the `ggplot2` R package.

7.1.1 Oscillations

After running the algorithm as described above with total mass $M = 0.25$, we find that the obtained clusterings and local density estimates are highly fluctuating over the iterations of the algorithm, as shown in figure 7.1:

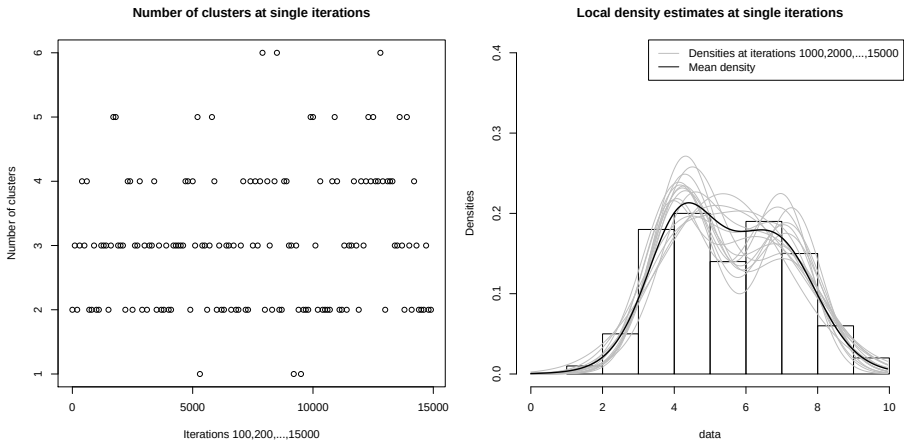


Figure 7.1: Clustering and density fluctuations over iterations

In both plots, a thinning of one iteration every 100 and every 2500, respectively, was performed for better readability of the plot. In the right side plot, the local densities are compared with the histogram of the data as well as the final estimate provided by the mean density. We can see that the number of clusters at all iterations varies significantly between 1 and 6, even in the last thousands of iterations, and the same behavior applies to the local density estimates. This is further confirmation of the fact that the single iterations themselves do not converge. Instead, as previously discussed, the convergence is in the *mean*, both for the density estimate and for the average dissimilarity matrix which we use to find the best clustering.

7.1.2 Total mass

Let us now examine the role of the total mass parameter, M. We ran the algorithm with several values for M whilst keeping the other parameters unchanged from the ones indicated at the beginning of the section. For each M, we saved the number of clusters of the best clustering produced by the algorithm, and studied the overall behavior varying M, shown in figure 7.2.

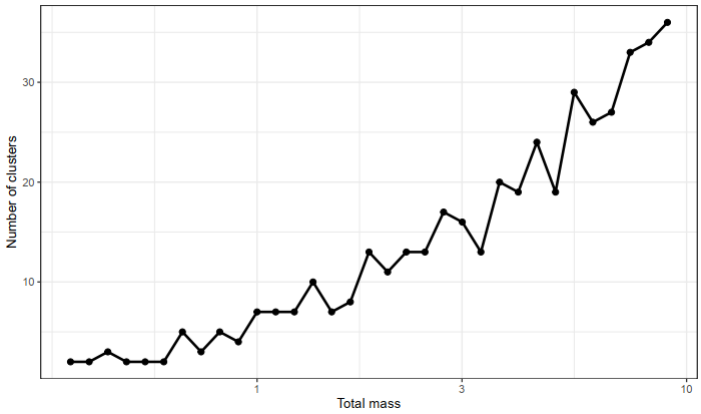


Figure 7.2: Number of clusters as a function of the total mass

Note that the values for M were chosen so as to be evenly spaced in log-scale, thus the abscissa is in log-scale as well. We can note that the clusters are increasing with the total mass. This is consistent with the fact that the probability of creating a new cluster is proportional to M, as seen in (2.4).
Moreover, we can see in figure 7.3 the density estimates for some of the values of M (again, compared with the histogram of the data).

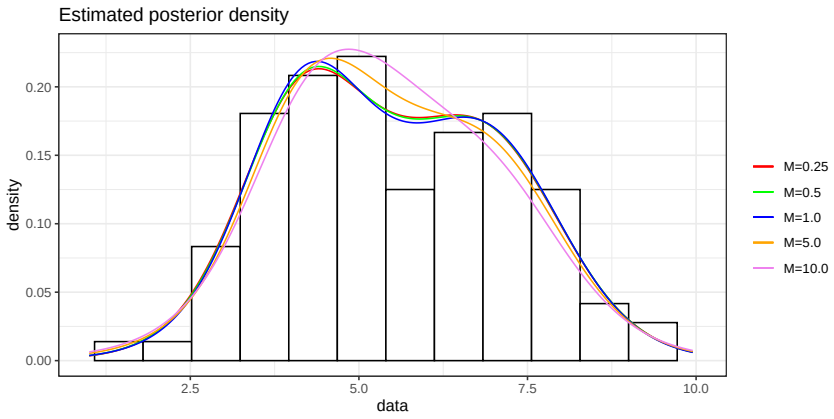


Figure 7.3: Density estimates varying the total mass

In our case, lower values for the total mass account better for the distribution of the data points, with the modes being near the real expected values of the two normal distributions, 4 and 7. On the other hand, higher values tend to clump together all 100 observations as though they were extracted from a single distribution. As we can see, the total mass M acts as smoothing parameter and, given its strong influence on the number of mixture components, it is a prime candidate for a prior distribution being put onto it.

7.1.3 Auxiliary blocks

We shall now try and change the number of auxiliary blocks m, and check how this impacts the density estimation. For this test, a large total mass $M = 10$ was chosen; the reason being that a small M would not allow significant differences as m changes. Indeed, m directly influences only the estimate (3.2) of the marginal distribution, that has a weight of $\frac{M}{M+n}$ (as seen in (3.1)), which is negligible if M is small. Therefore, $M = 10$ was picked, and the result is displayed in figure 7.4.

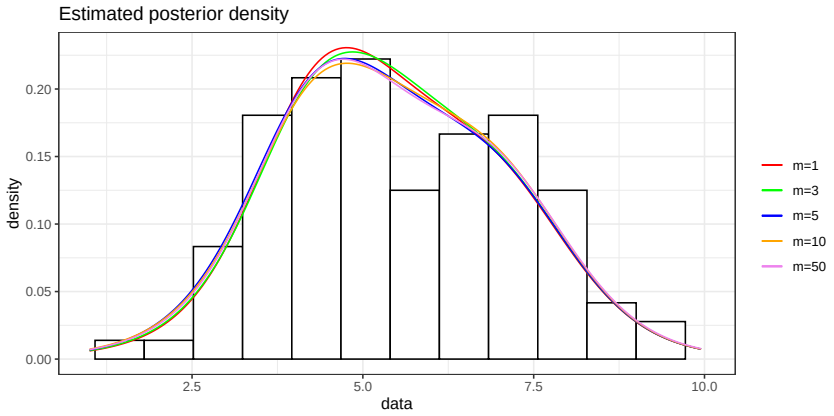


Figure 7.4: Density estimates varying the number of auxiliary blocks

Note that a larger m gives a better estimate of the marginal, because the sample mean is computed over a larger number of terms and the algorithm approximates the behavior of the algorithm `Neal2`.

7.1.4 Density components

We now wish to visualize how the local density is computed at a given sample iteration. Let us run again the `Neal8` algorithm with both $M = 0.25$ and $m = 3$ fixed, and then use the `cluster_estimate()` function to extract the best clustering for the data. We find that it is at iteration 2490, which gives 2 clusters. As shown in 3.1, each of these clusters has its own density estimate, which we refer to as *component*, and a weight attached to it proportional to its cardinality. The weighted sum of these components gives the "full" local estimate of the density for that iteration. The plot in figure 7.5 shows both the *weighted* components and their sum.

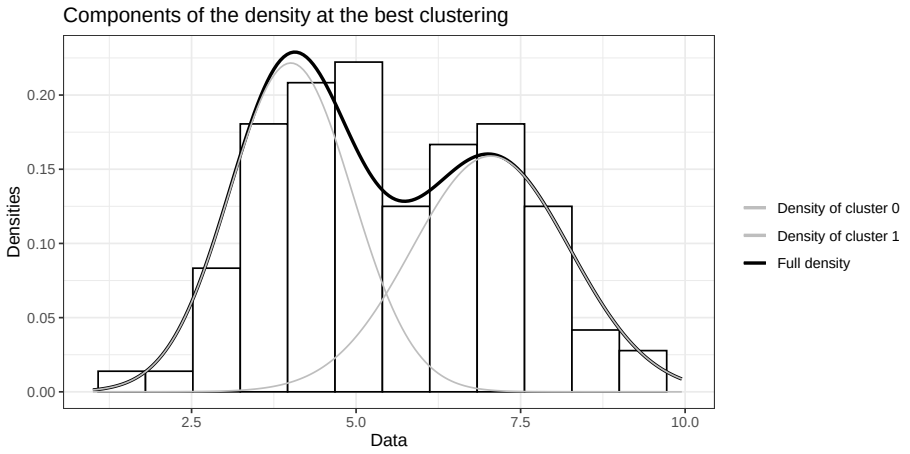


Figure 7.5: Weighted components and full density estimates

In this case the weights turn out to be approximately equal (0.52 and 0.48 respectively). Again, the two components are concentrated around the true means (4 and 7) of the likelihoods of the data points, as expected.
In other cases, the best clustering may produce more than 2 clusters. One such example is given by the best clustering of `Neal8` run with $M = 1$ (and $m = 3$ as before), found at iteration 6611. Although there are 7 clusters, all weights bar the first two are insignificant, as we can see in figure 7.6, making the corresponding components have almost zero impact in the weighted sum of the local estimate.

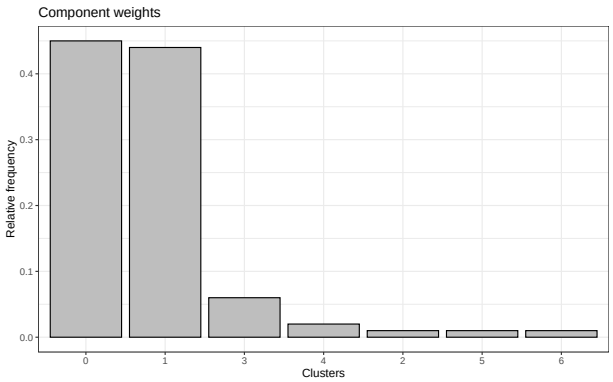


Figure 7.6: Clusters weights

7.1.5 `Neal2` vs `Neal8`

Finally, we ran the `Neal2` algorithm with the same parameters as `Neal8` (indicated at the beginning of the section) as well as $M = 10$ for both. Again, a rather large total mass was chosen in order to better highlight the difference in the marginal estimate. In fact, in the `Neal2` case, since the marginal distribution is known in closed form, the estimate is more accurate. A qualitative analysis is shown in figure 7.7.

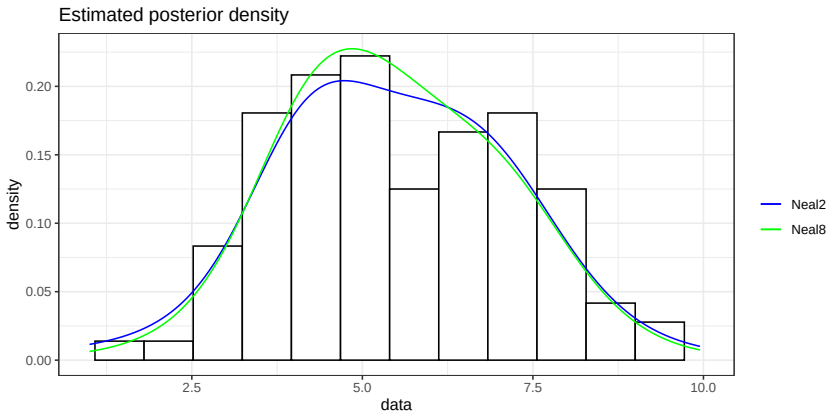


Figure 7.7: Neal2 and Neal8 density estimates

7.2 Python tests

We also implemented 6 tests in Python using data coming from different distributions and mixtures to assess the predicting power of the library. All of them were run with the following settings:

- `Neal2` algorithm with 500 iterations, of which 100 for the burn-in phase
- Dirichlet process mixture with total mass $m = 1$
- NNIG hierarchy for the univariate tests and the NNW hierarchy for the multivariate ones
- Hyperparameters for all NNIG hierarchies: $\mu_0 = 0.0$, $\lambda = 0.1$, $\alpha = 2$, $\beta = 2$
- Hyperparameters for all d-dimensional NNW hierarchies: μ_0 is the sample mean of the data, $\lambda = 0.2$, $\nu = d + 3$, $\tau_0 = \frac{1}{\nu} I_d$ (multiple of identity matrix).

Data were iid generated through the use of an R script. Below are listed the first 4 univariate tests alongside their respective sample size and the process from which the data was produced:

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	$y \sim 0.5\,t_5(-5, 1) + 0.5\,\text{SkewNormal}(5, 1, 2)$

Test number 4 uses a mixture of the Student's t distribution (refer to 1.4 for more information) and the skew normal distribution (location $\mu = 5$, scale $\sigma^2 = 1$, shape $\alpha = 2$). [1] Both distributions have similar shapes compared to a Gaussian p.d.f.; in particular, the Student's t converges to the Gaussian one as the degrees of freedom increase, while the skew normal is a generalization which is slightly shifted and tilted on one side, based on the shape parameter (for $\alpha > 0$, the distribution moves to the right and is tilted towards the left).
Tests 5 and 6 use multivariate data of increasing dimension, $d = 2$ and $d = 5$ respectively; for both of them, the sample size is $n = 400$ and data is generated from

$$y \sim 0.5\,\mathcal{N}(-3 \cdot 1_d, I_d) + 0.5\,\mathcal{N}(3 \cdot 1_d, I_d)$$

where 1_d is the d-dimensional unit vector and I_d the d-dimensional identity matrix.
For each univariate test, a plot was produced that includes the following:

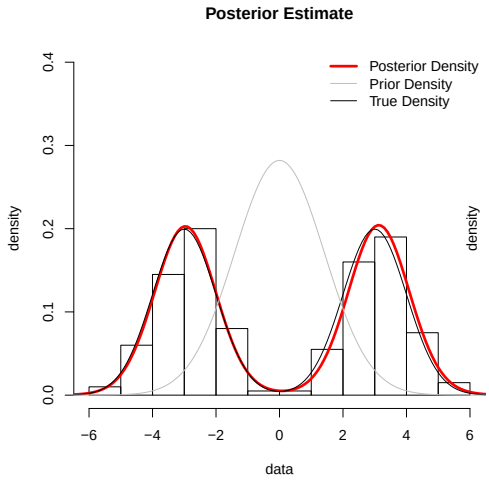
- the "expected prior likelihood" for the data, i.e. the Gaussian likelihood $\mathcal{N}(\mu, \sigma^2)$ with μ and σ^2 equal to the expected values provided by their own prior distribution;

[1] Its p.d.f. is as follows ($\phi(t) = (2\pi)^{-1/2} \exp\{-t^2/2\}$ is the standard $\mathcal{N}(0, 1)$ p.d.f.):

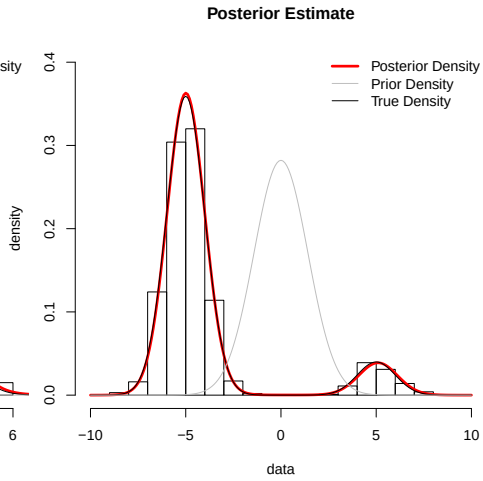
$$f(x) = 2\phi\left(\frac{x-\mu}{\sigma^2}\right) \int_{-\infty}^{\alpha \frac{x-\mu}{\sigma^2}} \phi(t)\mathrm{d}t$$

- the histogram produced by the actual data points used for the test;
- the true likelihood from which the data were extracted, as listed in the above table;
- the posterior likelihood computed by the library.

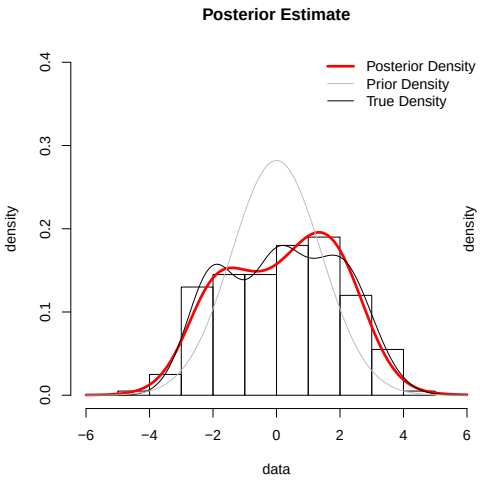
Having the "expected prior likelihood" as a rough starting point, the goal is to approximate the true likelihood via the posterior likelihood by using the given data. For multivariate tests, we show the following plots, obviously only for the $d = 2$ case: posterior likelihood and true likelihood, both in graph and in contour form.
Finally, the adjusted Rand index ARI for the predicted clustering with respect to the true clustering introduced in chapter 5 is also listed for all tests. The results are listed below.
The only test the library struggles with is number 3, in which only 2 clusters are recognized instead of the actual 3. This is due to the fact that the 3 clusters are really close to each other, therefore data for the central one can easily be mistaken as being part of the leftmost and rightmost ones.



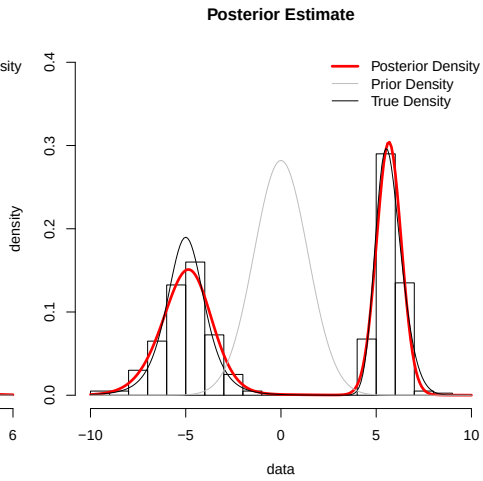
Test 1. $ARI = 1.0$



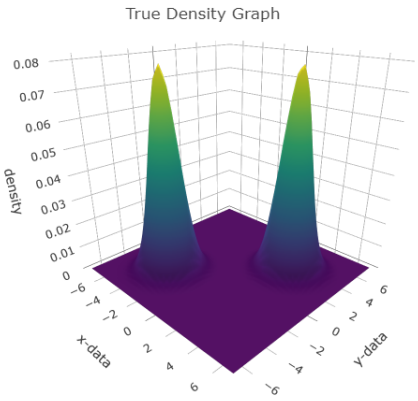
Test 2. $ARI = 1.0$

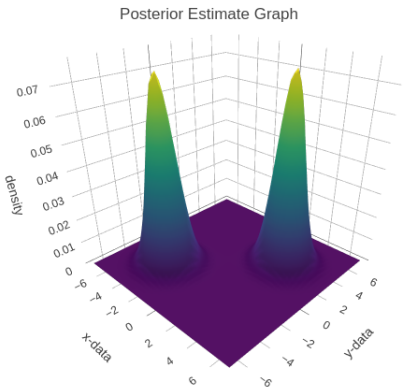


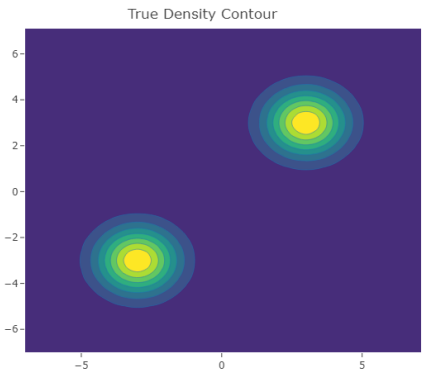
Test 3. $ARI = 0.45$

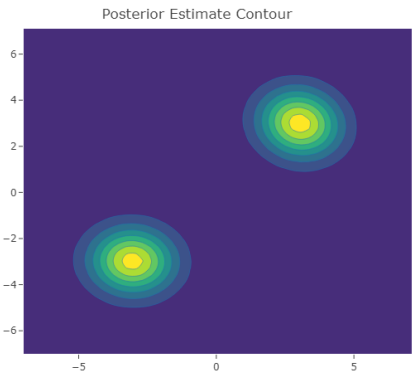


Test 4. $ARI = 0.99$









Test 5. $ARI = 1.0$

Chapter 8

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 implementing the currently empty extra step such as `update_hypers()`. 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 NNIG and NNW can be created. This is enough to run `Neal8` and `Neal2` by passing the class name as parameter, provided that the `Hierarchy` class has the appropriate interface.
- The same goes for other mixtures and algorithm. A good starting point for the latter would be the blocked Gibbs algorithm, which would make use of the `update_weights()` substep.
- The reading implementation for `FileCollector` without using the `get_chain()` method can be completed, as mentioned in section 4.4.
- Conjugacy-dependent algorithms such as `Neal2` can be further re-adapted to account for non-conjugacy, for example by using an Hamiltonian Monte Carlo sampler.
- Code parallelization can be studied, in particular for the `cluster_estimate()` function, in which dissimilarity matrices can be computed in parallel. Note that the algorithm themselves are sequential by nature, so there is not much else to optimize this way.
- 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. Although it is a hard task, we do think it is possible to achieve with reasonable effort.

Bibliography

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

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

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

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

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

[6] Rand W. (1971), *Objective Criteria for the Evaluation of Clustering Methods. Journal of the American Statistical Association*, 66(336), 846-850. doi:10.2307/2284239

[7] Stan documentation: `http://mc-stan.org/math`.
Code found at `https://github.com/stan-dev/math` (current version 3.2.0 was used) and it also includes other needed libraries: Boost, Eigen, SUNDIALS, Intel TBB

[8] Eigen documentation: `https://eigen.tuxfamily.org/dox`.
Code is included in the Stan package (current version 3.3.7 was used)

[9] 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)

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

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

[12] 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

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

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