Journal of Statistical Software
March 2025, Volume 112, Issue 8. *doi: 10.18637/jss.v112.i08*

pyrichlet: A Python Package for Density Estimation and Clustering Using Gaussian Mixture Models

Fidel Selva
Uiversidad Nacional Autónoma de México

Ruth Fuentes-García
Uiversidad Nacional Autónoma de México

María Fernanda Gil-Leyva
Uiversidad Nacional Autónoma de México

Abstract

Bayesian nonparametric models have proven to be successful tools for clustering and density estimation. While there exists a nourished ecosystem of implementations in R, for Python there are only a few. Here we develop a Python package called **pyrichlet**, for Bayesian nonparametric density estimation and clustering using various state-of-the-art Gaussian mixture models that generalize the well established Dirichlet process mixture, many of which are fairly new. Implementation is performed using Markov chain Monte Carlo techniques as well as variational Bayes methods. This article contains a detailed description of **pyrichlet** and examples for its usage with a real dataset.

Keywords: density estimation, clustering, random partitions, Gaussian mixture, Dirichlet process, geometric process, Pitman-Yor, Python.

1. Introduction

Bayesian nonparametric (BNP) methods provide versatile and elegant solutions for modeling complex data structures where parametric methods do not provide sufficient flexibility (Hjort, Holmes, Müller, and Walker 2010). In particular, Gaussian mixture models (GMM) represent one of the most successful applications of BNP methods for clustering and density estimation. Broadly speaking, GMM consist of components identified by a unique Gaussian distribution and a mixing proportion or weight. Each observation is then assumed to be drawn from one of these Gaussian distributions with probability determined by the corresponding weight. Frequentist inference usually proceeds by maximizing the likelihood (Xu and Jordan 1996;

Pernkopf and Bouchaffra 2005; Vila and Schniter 2013), in contrast Bayesian methods assign a prior to the mixing distribution, which is a discrete random probability measure whose atoms and their sizes represent the component parameters and weights, respectively. A natural choice to attain conditional conjugacy is to assign a Normal-Inverse Gamma or Normal-Inverse Wishart prior to the Gaussian parameters, as for the weights the semi-parametric approach commonly assumes the number of components is known and assigns a Dirichlet prior to the weighting structure (e.g., Ishwaran and James 2000). Instead, BNP methods consider mixing distributions with infinitely many number of support points and infer about the number of components through the number of clusters of data points. This plays a key role in the success of BNP methods, as the number of components is rarely known in advance. The canonical example of BNP Gaussian mixtures is the Dirichlet process mixture (DPM) (Ferguson 1983; Lo 1984). Other well-known examples are the Pitman-Yor process mixture (PYM) (Pitman and Yor 1997; Ishwaran and James 2001), which generalizes the DPM, and the geometric process mixture (GPM) introduced by Fuentes-García, Mena, and Walker (2010). Although the DPM and PYM models are very good at fitting complex weighting structures, which makes them appealing for clustering purposes, the GPM is faster for density fitting. Recently, generalizations and interpolations of DPM and GPM have been have been proposed (cf. Gil–Leyva, Mena, and Nicoleris 2020; Gil-Leyva and Mena 2021).

While BNP models provide an increased flexibility, they are also more challenging to implement as one has to deal with an infinite model dimension. One approach to perform posterior inference is to design a Markov chain Monte Carlo (MCMC) method aimed at drawing samples from the posterior distribution. For the DPM and PYM there are various Gibbs sampler variants that can be used, in particular the so-called marginal methods (e.g., Escobar and West 1995; Neal 2000) exploit the generalized Pólya urn scheme representation of the model (Blackwell and MacQueen 1973; Pitman 2006). Unfortunately, this representation is not available for most prior choices such as the ones introduced by Fuentes-García *et al.* (2010); Gil–Leyva *et al.* (2020); Gil-Leyva and Mena (2021). Alternatively, one can rely on a conditional Gibbs sampler, such as the Slice sampler by Walker (2007); Kalli, Griffin, and Walker (2011), which is more widely implementable and has important computational advantages in big-data settings (Gelfand and Kottas 2002). A different approach to perform posterior inference is offered by variational Bayes (VB) methods which approximate the posterior distribution using simplified distributions with a tractable form (Blei and Jordan 2006). The package **pyrichlet** implements all the considered BNP and finite models by means of a Slice sampler; additionally, many also include a VB implementation.

There is a nourished ecosystem of packages written for the statistical programming language R which can be used to do density estimation or clustering but only a few of them implement BNP models. Some examples are **BNPmix** (Corradin, Canale, and Nipoti 2021) mainly written in C++ which has density estimation and clustering capabilities by using the DPM and PYM models. In **mclust** (Scrucca, Fop, Murphy, and Raftery 2016) the expectation-maximization (EM) algorithm is used to fit a finite GMM with a varying number of components and with optimizations written in Fortran. The **dirichletprocess** package (Ross and Markwick 2020) can be used to implement multivariate Gaussian mixtures under a DPM model using MCMC. Also **bayesm** (Rossi 2019), which is optimized in C++, has an MCMC implementation for the Dirichlet Distribution Mixture (DDM) model. Regarding **DPpackage** (Jara, Hanson, Quintana, Mueller, and Rosner 2011), despite its broad usage and its implementations for several BNP models based on the DP, it has been archived due to system

	Methods		Models	
Packages	MCMC	Variational/EM	Finite GMM	BNP GMM
R packages				
BNPmix	✓	×	×	✓
mclust	×	✓	✓	×
dirichletprocess	✓	×	×	✓
bayesm	✓	×	×	✓
DPpackage	✓	×	×	✓
PReMiuM	✓	×	✓	✓
BayesMixR	✓	×	✓	✓
Python packages				
scikit-learn	×	✓	✓	✓
mixes	×	✓	✓	×
Mixture-Models	×	✓	✓	×
BayesMixPy	✓	×	✓	✓
pyrichlet	✓	✓	✓	✓

Table 1: Comparative table of R and Python packages

I/O calls coded directly in Fortran. Lastly for R, the package **PReMiuM** (Liverani, Hastie, Azizi, Papathomas, and Richardson 2015) was developed to carry out mainly response-covariate profile regressions with Gaussian, discrete or mixed covariate data, and a binary, categorical, count or continuous response by fitting a DPM model; although users can also choose to exclude the response variable from the model, thus returning to a non-response DPM model.

For Python there are not many packages with features that are comparable to those of R, being **scikit-learn** (Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg, Vanderplas, Passos, Cournapeau, Brucher, Perrot, and Duchesnay 2011), to the best of our knowledge, the only native and well documented library for performing density estimation and clustering using BNP Gaussian mixtures. The packages **mixes** citepDmytruk2022 and **Mixture-Models** (Kasa 2022; Kasa and Rajan 2020) can be used to do density estimation and clustering by maximizing the likelihood of a finite Gaussian mixture using EM and gradient descent with automatic differentiation respectively. Additionally, the C++ library **BayesMix** (Beraha, Guindani, Gianella, and Guglielmi 2025) can be used to fit finite and BNP mixtures using several of the MCMC algorithms described in Neal (2000), and can be used in Python and R via the wrapper packages **BayesMixPy** (Beraha, Guindani, Gianella, and Guglielmi 2023a) and **BayesMixR** (Beraha, Guindani, Gianella, and Guglielmi 2023b) respectively. In Table 1 we summarize the discussed packages.

2. Gaussian mixture models

2.1. Model specification

In mixture models we assume that data points, $\mathbf{y} = \{y_1, \dots, y_N\}$ with values in $\mathbb{R}^p$, are independent and identically distributed from a probability measure, Q, obtained by mixing a parametric family of distributions, $\{K_\theta : \theta \in \Theta\}$, with respect to a discrete probability measure, P:

$$\mathsf{Q}\,(\cdot) := \int K_\theta\,(\cdot)\ \mathsf{P}(\mathrm{d}\theta).$$

In particular, if $K_\theta = \mathcal{N}(\,\cdot \mid \mu, \Sigma)$ stands for a multivariate Normal distribution over $(\mathbb{R}^p, \mathcal{B}_{\mathbb{R}^p})$, with $\theta = (\mu, \Sigma)$, then Q is termed a Gaussian mixture. In the Bayesian framework, the *mixing distribution*, P, is considered to be random, and to fully specify the model it is enough to assign a prior distribution to P. The most famous examples of mixture models feature a mixing distribution that belongs to the general class of *proper species sampling models* introduced and studied by Pitman (1996). These are discrete random probability measures over a Borel space, $(\Theta, \mathcal{B}_\Theta)$,

$$\mathsf{P}\,(\cdot) = \sum_{j=1}^{\infty} w_j \delta_{\theta_j}(\cdot), \tag{1}$$

where the weighting structure, $\mathbf{w} := \{w_j\}_{j=1}^\infty$, is independent of the atoms structure, $\boldsymbol{\theta} := \{\theta_j\}_{j=1}^\infty$, and the entries of $\boldsymbol{\theta}$ are iid from a diffuse distribution G over $(\Theta, \mathcal{B}_\Theta)$. When P is as in (1) we can rewrite the mixture model as $\mathsf{Q}(\cdot) = \sum_{j=1}^\infty w_j\, K_{\theta_j}(\cdot)$, in this case to specify the prior distribution of P, it is enough to choose G and assign a prior to $\mathbf{w}$. Here, in order to attain conjugacy between G and the Gaussian kernel K_θ, we consider G to be a Normal-Inverse Gamma distribution for univariate data points and a Normal-Inverse Wishart distribution for multivariate datasets. As for $\mathbf{w}$, specifying its law is a more delicate issue. First of all, it should be noticed that this sequence must take values in the simplex,

$$\Delta_\infty = \left\{(s_1, s_2, \dots)\ : s_j \geq 0,\ \textstyle\sum_{j=1}^\infty s_j = 1\right\},$$

in order for P to be a well-defined probability measure, and it is not straight-forward to define distributions supported on Δ_∞. Secondly, as explained by Bissiri and Ongaro (2014), it is an essential requirement for posterior consistency that the weighting structure defines a proper species sampling process with full support. Loosely speaking it is required that the (weak topological) support of P is the biggest possible (only constrained by the set of probability measures, μ, whose support is contained in the support G). For clustering estimation no such requirement exists since BNP models such as the DPM and PYPM do not guarantee consistency (Miller and Harrison 2014). A third reason of why choosing the prior of $\mathbf{w}$ must be taken with care is that it determines the prior behavior of the clustering of data points. To explain this, first note that in mixture models we can equivalently assume that the data points are independently sampled from $y_i \mid x_i \overset{ind}{\sim} K_{x_i}(\cdot)$, where $\{x_i\}_{i=1}^N$ are conditionally iid from P given P. Under this statement of the model we can introduce the *allocation variables*, $\mathbf{d} = \{d_i\}_{i=1}^N$, given by $d_i = j$ if and only if $x_i = \theta_j$, and define the equivalence relation $i \sim_{\mathbf{d}} l$ if and only if $d_i = d_l$. This way $\mathbf{d}$ induces a clustering of the (indexes of) observations according to which component of the mixture they were sampled from, and it holds that $d_i \mid \mathbf{w} \overset{iid}{\sim} \sum_{j=1}^\infty w_j \delta_j$, i.e., $\mathbb{P}[d_i = j \mid \mathbf{w}] = w_j$. Denoting by Π_N the partition of $\{1, \dots, N\}$ induced by $\sim_{\mathbf{d}}$ we get

$$\mathbb{P}[\Pi_N = \{A_1, \dots, A_k\}] = \sum_{(j_1, \dots, j_k)} \mathbb{E}\left[\prod_{l=1}^{k} w_{j_l}^{|A_l|}\right], \tag{2}$$

for every partition $\{A_1, \dots, A_k\}$ of $\{1, \dots, N\}$, where $|A_l|$ denotes the cardinality of A_l, and the sum ranges over all k-tuples, $(j_1, \dots, j_k)$, of distinct positive integers (cf. Pitman 2006).

It is then clear by (2) that the prior distribution of the clustering structure is completely determined by the law of $\mathbf{w}$. However it is not easy to understand in detail how will the latter affect the clustering estimation. For this reason **pyrichlet** offers a wide variety of prior choices for $\mathbf{w}$, among which the user can select. All the BNP models considered here are well-defined, i.e., $\sum_{j=1}^{\infty} w_j = 1$; all feature a mixing prior with full support, and they enjoy a simple stick-breaking decomposition, as described next.

Stick-breaking weighting structures

The stick-breaking decomposition (Sethuraman 1994; Ishwaran and James 2001) translates the problem of defining a distribution supported in Δ_∞, into the simpler one of defining the law of a sequence of *sticks*, $\mathbf{v} = \{v_j\}_{j=1}^{\infty}$, by decomposing

$$w_1 = v_1, \quad w_j = v_j \prod_{k=1}^{j-1}(1 - v_k), \quad j \geq 2. \tag{3}$$

To assure that the mapping

$$\mathcal{SB}(\mathbf{v}) = \mathcal{SB}(v_1, v_2, \ldots) \mapsto (v_1, v_1(1 - v_2), \ldots) = \mathbf{w}$$

defines a valid weighting structure it is enough to require $\sum_{j=1}^{\infty} v_j = \infty$ or $v_j = 1$ for some $j \geq 1$. Indeed, noticing that $\sum_{j=1}^{n} w_j = 1 - \prod_{j=1}^{n}(1 - v_j)$, it is clear that $\sum_{j=1}^{\infty} w_j = 1$ if and only if $\lim_{n\to\infty} \prod_{j=1}^{n}(1 - v_j) = 0$, which in turn occurs when $\sum_{j=1}^{\infty} v_j = \infty$ or $v_j = 1$ for some $j \geq 1$. Additionally, Bissiri and Ongaro (2014) showed that, in terms of the sticks, it is enough to require that for every $\varepsilon > 0$ there exist $0 < \gamma < \varepsilon$ such that

$$\mathbb{P}[\gamma < v_j < \varepsilon, \text{ for all } j \in \{1, \ldots, n\}] > 0$$

for each $n \geq 1$, in order for P to have full support.

The canonical example of a species sampling processes, P, in BNP statistics is the *Dirichlet Process* (Ferguson 1973; Sethuraman 1994) with total mass parameter $b > 0$. It arises when the sticks, $\mathbf{v}$, are iid from a $\mathrm{Beta}(1, b)$ distribution ($v_j \overset{iid}{\sim} \mathrm{Beta}(1, b)$). The Dirichlet process is a very tractable model and various distinct representations are available (cf. Ferguson 1973; Regazzini, Lijoi, and Prünster 2003; Blackwell and MacQueen 1973; Pitman 2006; Hjort *et al.* 2010), which explains why it plays such an important role in the BNP literature.

Relaxing the identical distribution assumption and keeping the independence on $\mathbf{v}$ elements, we find the two-parameter *Pitman-Yor Process* (Pitman and Yor 1997; Ishwaran and James 2001), where $v_j \overset{ind}{\sim} \mathrm{Beta}(1 - a, b + ja)$. Evidently the choice $a = 0$ recovers a Dirichlet process. In contrast to the weights of Dirichlet processes, Pitman-Yor weights decay at a slower rate and thus PYPM are more prone to estimate a larger number of clusters of data points, according to the mixture component they were sampled from.

Another direction to define different stick-breaking processes is to keep the identical distribution assumption and relax the independence one. In this scenario we find the *Geometric Process*, first introduced by Fuentes-García *et al.* (2010), and obtained by setting $v_j = v$ for every $j \geq 1$ and some random variable $v \sim \mathrm{Beta}(a, b)$, so that the weights become $w_j = v(1 - v)^{j-1}$. In contrast to DPM and PYPM, GPM tends to perform worse for clustering purposes, nonetheless, it has been observed that for density estimation they can capture details that some DPM struggle to do. This has recently motivated the study stick-breaking species sampling processes, P, where $\mathbf{v}$ forms an exchangeable sequence (Gil-Leyva and Mena 2021) or a Markov process (cf. Gil–Leyva *et al.* 2020), as both classes generalize Dirichlet and geometric processes simultaneously. In particular, (Gil-Leyva and Mena 2021) studied the general case where $\mathbf{v}$ is exchangeable, and gave sufficient conditions so that P is well-defined and has full support. Following their work, here we consider a model where $v_j \mid \mathsf{P}' \overset{iid}{\sim} \mathsf{P}'$, and P' is a Dirichlet process over $([0,1], \mathcal{B}_{[0,1]})$ with atoms distribution $G' = \text{Beta}(a,b)$ and total mass parameter b'. In this case we term the main species sampling process, P over $(\Theta, \mathcal{B}_\Theta)$ a *Beta in Dirichlet Process* with parameters $a, b, b' > 0$. As proven by Gil-Leyva and Mena (2021), as $b' \to 0$, P approximates in distribution a Geometric process; and if $a = 1$, as $b' \to \infty$, P approximates in distribution a Dirichlet process with total mass parameter b. Additionally, we will consider a second exchangeable stick-breaking process, where given $v \sim \text{Beta}(a,b)$ the sticks satisfy

$$v_j \mid v \overset{iid}{\sim} \text{Beta}\left(a + \frac{x}{1-x}v, b - \frac{x}{1-x}(1-v)\right),$$

with $a, b > 0$ and $0 \leq x < 1$. In this case we call the species process, P, a *Beta in Beta Process*. Note that the choice $x = 0$ recovers a Dirichlet process and as $x \to 1$, P approximates weakly a Geometric process Selva (2020). On the Markov counterpart, we have implemented a *Beta-Binomial Process*, P, introduced and studied by Gil–Leyva *et al.* (2020). For this model the law of the sticks can be described through

$$B_j \mid v_{j-1} \sim \text{Binomial}(n, v_{j-1}), \quad v_j \mid B_j \sim \text{Beta}(a + B_j, b + (n - B_j)),$$

where $v_0 \sim \text{Beta}(a,b)$, $a, b > 0$ and $n \in \mathbb{Z}_+$. Thus, it follows from the Beta-Binomial conjugate model that after marginalizing out $\{B_j\}_{j=1}^{\infty}$, we obtain a Markov chain, $\mathbf{v}$, with stationary distribution $\text{Beta}(a,b)$. Moreover, under the convention $\text{Binomial}(0,p) = \delta_0$, for the choice $n = 0$, P becomes a Dirichlet process, and as proven by Gil–Leyva *et al.* (2020), as $n \to \infty$, P converges in distribution to a Geometric process.

The **pyrichlet** package also implements some models that do not lay in the BNP realm. In particular, we have considered the Dirichlet Distribution Mixture: a Bayesian (semi-parametric) model whose mixing distribution, P, has a finite number, $n \in \mathbb{Z}_+$, of support points (i.e., $w_j = 0$ for every $j > n$); and Dirchlet distributed weights $(w_1, \ldots, w_n) \sim \text{Dir}(\boldsymbol{\alpha})$, where the concentration parameter is defined by $\boldsymbol{\alpha} := (\alpha_1, \ldots, \alpha_n)$. Unfortunately, P does not have full support in this case, nonetheless, if $\alpha_j = b/n$ for some $b > 0$, then by making $n \to \infty$, the distribution P resembles that of a Dirichlet process with total mass parameter $b > 0$ (Blackwell and MacQueen 1973). It means that if n is large enough, this model will behave similarly to one that does has a mixing prior with full support. The two other models proposed are limiting cases of the Dirichlet Distribution Mixture when the parameter α tends to zero and to infinity respectively. For this three models an exploratory analysis over the number of components can be carried as in McKenzie and Alder (1994), and, depending on the length selection method, consistency can be regained.

2.2. Implementation methods

As mentioned in Section 2.1 we can describe the mixture model through

$$y_i \mid (\boldsymbol{\theta}, d_i) \overset{ind}{\sim} K_{\theta_{d_i}}, \quad d_i \mid \mathbf{w} \overset{iid}{\sim} \sum_{j=1}^{\infty} w_j \delta_j, \quad \theta_j \overset{iid}{\sim} G, \quad \mathbf{w} \sim f(\mathbf{w}) \tag{4}$$

where $f(\mathbf{w})$ refers to the prior law of the weighting structure. This means that the joint distribution factorizes as

$$\begin{aligned} f(\mathbf{y}, \mathbf{d}, \boldsymbol{\theta}, \mathbf{w}) &= \left(\prod_{i=1}^{N} f(y_i \mid \theta_{d_i}) f(d_i \mid \mathbf{w}) \right) f(\boldsymbol{\theta}) f(\mathbf{w}) \\ &= \left(\prod_{i=1}^{N} w_{d_i} K_{\theta_{d_i}}(y_i) \right) f(\boldsymbol{\theta}) f(\mathbf{w}), \end{aligned} \tag{5}$$

assuming $K_{\theta_{d_i}}$ has a density, denoted by the same symbol, with respect to the Lebesgue measure. In general the posterior distribution, $f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y}) \propto f(\mathbf{y}, \mathbf{d}, \boldsymbol{\theta}, \mathbf{w})$, can not be computed explicitly, and BNP models have the additional challenge of having an infinite dimensional support. To make their fitting possible, various methods have been derived. In particular MCMC techniques, such as Gibbs samplers (cf. Escobar and West 1995; Neal 2000; Walker 2007; Kalli *et al.* 2011) consist in constructing a stationary Markov chain whose invariant measure is $f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})$. This way, after choosing a suitable initialization point and allowing the chain to evolve long enough, one can obtain samples from $f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})$ and estimate quantities of interest through measurable functions of the sampled variables. Instead VB techniques (Jordan, Ghahramani, Jaakkola, and Saul 1999; Opper and Saad 2001; Blei and Jordan 2006) approximate the posterior distribution $f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})$ by means of a distribution $g \in \mathcal{G}$, that minimizes the Kullback-Leiber divergence among a class of proposals, $\mathcal{G}$, with a simplified tractable form. Then quantities of interest can be estimated through functionals of the variational distribution g. Next we describe the MCMC and VB variants available in **pyrichlet** for the implementation of Gaussian mixture models.

Gibbs sampling with slices

To motivate the Slice sampler, consider the mixture model description in (4) and assume that the model dimension, n, is a finite fixed number, i.e., $w_j > 0$ for $j \leq n$ and $w_j = 0$ for each $j > n$. In this setting a simple Gibbs sampling algorithm would consist in consecutively drawing samples from the full conditionals:

$$\begin{aligned} f(d_i \mid \cdots) &\propto K_{\theta_{d_i}}(y_i)\, w_{d_i}\, \mathbb{1}(d_i \in \{1, \ldots, n\}), \quad i \in \{1, \ldots, N\}, \\ f(\theta_j \mid \cdots) &\propto G(\theta_j) \prod_{i \in D_j} K_{\theta_j}(y_i), \quad j \in \{1, \ldots, n\}, \\ f(\mathbf{w} \mid \cdots) &\propto f(\mathbf{w}) \prod_{j=1}^{n} w_j^{|D_j|}, \end{aligned}$$

which are proportional to (5), where $D_j = \{i : d_i = j\}$. For example, if $f(\mathbf{w}) = \mathrm{Dir}(\mathbf{w} \mid \alpha_1, \ldots, \alpha_n)$ then $f(\mathbf{w} \mid \cdots) = \mathrm{Dir}(\mathbf{w} \mid \alpha_1 + |D_j|, \ldots, \alpha_n + |D_n|)$; and if G and K form a conjugate pair, then sampling from $f(\theta_j \mid \cdots)$ is easy. In particular, if K_{θ_j} is a Gaussian kernel $\mathcal{N}(\cdot \mid \mu_j, \Sigma_j)$ with $\theta_j = (\mu_j, \Sigma_j)$, and G stands for a Normal-Inverse Wishart distribution with hyperparameters $(m_0, \lambda_0, \Psi_0, \nu_0)$, we find $f(\theta_j \mid \cdots) = f(\mu_j, \Sigma_j \mid \cdots)$ is also a Normal-Inverse

Wishart distribution with updated parameters

$$\begin{aligned}\lambda_j &= \lambda_0 + n_j,\\ m_j &= \lambda_j^{-1}(\lambda_0 m_0 + n_j \overline{y}_j),\\ \Psi_j &= \Psi_0 + n_j S_j + \lambda_j^{-1}\lambda_0 n_j(\overline{y}_j - m_0)(\overline{y}_j - m_0)^\top,\\ \nu_j &= \nu_0 + n_j,\end{aligned}$$

where $n_j = |D_j|$, $\overline{y}_j = n_j^{-1}\sum_{i\in D_j} y_i$, $S_j = n_j^{-1}\sum_{i\in D_j}(y_i - \overline{y}_j)(y_i - \overline{y}_j)^\top$. Notice that if the model dimension n is infinite, the aforementioned Gibbs sampler is not feasible to implement as it is not possible to draw samples from the complete sequences $\boldsymbol{\theta}$ and $\mathbf{w}$, which are necessary for the subsequent update of $\mathbf{d}$. To overcome this, various Gibbs sampler variants have been derived (cf. Neal 2000), in particular Walker (2007) proposed to introduce latent variables, $\mathbf{u} = \{u_i\}_{i=1}^n$, yielding the augmented joint distribution

$$\begin{aligned} f(\mathbf{y},\mathbf{d},\mathbf{u},\boldsymbol{\theta},\mathbf{w}) &= \left(\prod_{i=1}^N f(y_i \mid \theta_{d_i}) f(d_i \mid u_i, \mathbf{w}) f(u_i \mid \mathbf{w})\right) f(\boldsymbol{\theta}) f(\mathbf{w})\\ &= \left(\prod_{i=1}^N K_{\theta_{d_i}}(y_i)\mathbb{1}\{w_{d_i} > u_i\}\right) f(\boldsymbol{\theta}) f(\mathbf{w}),\end{aligned} \tag{6}$$

with $f(d_i \mid \mathbf{w}, u_i) = \mathbb{1}\{w_{d_i} > u_i\}\left(\sum_{j=1}^\infty \mathbb{1}\{w_j > u_i\}\right)^{-1}$ and $f(u_i \mid \mathbf{w}) = \sum_{j=1}^\infty \mathbb{1}\{w_j > u_i\}$. Integrating out $\mathbf{u}$ from (6) one recovers (5), the advantage of working with the latter is that the problem of sampling infinitively many random variables is translated into that of sampling a random number of them.

Unlike other fitting methods, the slice sampler avoids truncating the infinite dimensional weighting structure, yielding an exact method that does not distort the model. However, this method can suffer from slow iteration times caused by the size of the sliced groups, $\sum_{j=1}^\infty \mathbb{1}\{w_j > \min u_i\}$, being too large. This in turn means that at some iterations a very large number of weights need to be updated, in which case a truncated model might be faster to fit. The sliced groups' size depends both on the number of observations and on the dynamics of the weighting model, but the weighting structure exerts a greater influence. This problem has been observed in the Pitman-Yor process for large values of the discount parameter (cf. Canale, Corradin, and Nipoti 2022) since the stick lengths v_j lean towards zero when the discount parameter is positive, translating in ever bigger. The rest of the models implemented in this package do not exhibit this behavior due to the stationarity of $\mathbf{v}$.

Following Kalli *et al.* (2011), **pyrichlet** implements the so called efficient-slice sampler that works with the following conditionals:

$$\begin{aligned} f(d_i \mid \cdots) &\propto K_{\theta_{d_i}}(y_i)\,\mathbb{1}\{w_{d_i} > u_i\} \quad i \in \{1,\dots,N\},\\ f(\theta_j \mid \cdots) &\propto G(\theta_j)\prod_{i\in D_j} K_{\theta_j}(y_i), \quad j \in \{1,\dots,n\},\\ f(\mathbf{u},\mathbf{w} \mid \cdots) &\propto f(\mathbf{w})\prod_{i=1}^N \mathbb{1}\{w_{d_i} > u_i\}.\end{aligned}$$

Here $f(\theta_j \mid \cdots)$ is identical as before, and $f(d_i \mid \cdots)$ is a discrete distribution with finite and non-empty support, $\{j : w_j > u_i\}$, hence for Gaussian mixture models it is easy to sample

Algorithm 1 Gibbs sampling with slices

```
procedure FIT_GIBBS(y)
    $N \leftarrow \text{length}(y)$
    Initialize $\{u_j\}_{j=1}^N$ with a Uniform(0, 1) distribution
    Draw $\{w_j\}_{1\leq j}$ from the prior weighting model until $\sum_{l=1}^j w_l > 1 - \min\{u_i\}_{i=1}^N$ holds
    $j^* \leftarrow \min\{j \mid \sum_{l=1}^j w_l > 1 - \min\{u_i\}_{i=1}^N\}$
    Initialize $\{d_j\}_{j=1}^N$ from the truncated weights $\{w_j\}_{1\leq j\leq j^*}$
    Initialize $\{\theta_j\}_{j\geq 1}$ from the prior distribution of atoms
    for $s$ in $1 : total\_iter$ do
        Update existing atoms $\boldsymbol{\theta}$ with the full conditional $f(\boldsymbol{\theta} \mid \mathbf{y}, \mathbf{u}, \mathbf{d}, \mathbf{w})$
        Update existing weights $\mathbf{w}$ with the full conditional $f(\mathbf{w} \mid \mathbf{y}, \mathbf{u}, \mathbf{d}, \boldsymbol{\theta})$ or $f(\mathbf{w} \mid \mathbf{y}, \mathbf{d}, \boldsymbol{\theta})$
        Update $\mathbf{u}$ with the full conditional $f(\mathbf{u} \mid \mathbf{y}, \mathbf{d}, \mathbf{w}, \boldsymbol{\theta})$
        $j^* \leftarrow \min\{j \mid \sum_{l=1}^j w_l > 1 - \min\{u_i\}_{i=1}^N\}$
        Complete any missing weights up to $j^*$ with the prior weighting model
        Complete any missing atoms up to $j^*$ with the prior atom model
        Update $\mathbf{d}$ with the full conditional $f(\mathbf{d} \mid \mathbf{y}, \mathbf{u}, \mathbf{d}, \mathbf{w}, \boldsymbol{\theta})$
        Save state variables $(\mathbf{u}_s^*, \mathbf{d}_s^*, \mathbf{w}_s^*, \boldsymbol{\theta}_s^*) = (\mathbf{u}, \mathbf{d}, \mathbf{w}, \boldsymbol{\theta})$
    end for
end procedure
```

from both distributions. As for $f(\mathbf{u}, \mathbf{w} \mid \cdots)$, we can first sample $\mathbf{w}$ from

$$f(\mathbf{w} \mid \cdots - \mathbf{u}) \propto f(\mathbf{w}) \prod_{j=1}^{\max \mathbf{d}} w_j^{|D_j|}, \tag{7}$$

which is obtained after integrating $f(\mathbf{u}, \mathbf{w} \mid \cdots)$ with respect to $\mathbf{u}$, and later sampling $\mathbf{u}$ from

$$f(\mathbf{u} \mid \cdots) \propto \prod_{i=1}^{N} \mathbb{1}\{w_{d_i} > u_i\},$$

a product of independent Uniform$(0, w_{d_i})$ distributions. At this stage it only remains to explain how to sample from (7). If the stick-breaking decomposition (3) is available, this can be done via sampling the sticks, $\mathbf{v}$, from

$$f(\mathbf{w} \mid \cdots - \mathbf{u}) \propto f(\mathbf{v}) \prod_{j=1}^{\max \mathbf{d}} v_j^{a_i} (1 - b_j)^{b_j}, \tag{8}$$

with $a_j = |D_j|$ and $b_j = \sum_{i>j} |D_i|$. For example, if $v_j \overset{iid}{\sim} \text{Beta}(1, b)$, we update the sticks by independently sampling $v_j \sim \text{Beta}(1 + a_j, b + b_j)$.

For the remaining weighting structures, details on how to sample from (8) are provided in Appendix A. It is worth highlighting that it is necessary to sample v_j and θ_j for every $j \geq 1$, and it is enough to sample them up to the point where the updating algorithm for $\mathbf{d}$ can take place, that is up to the first index J such that $\sum_{j=1}^J w_j \geq \max_{i\leq N}(1 - u_i)$, after that it is not possible that $w_j > u_i$ for some $j > J$ and $i \leq N$.

In Algorithm 1 the complete procedure to draw samples of the model's variables is given, this variables allow us to estimate any quantity of interest if it can be derived from the posterior distribution of the variables. For example, if we wish to compute the posterior distribution $f(y \mid \mathbf{y})$, we can estimate it by recalling the saved state variables $\{(\mathbf{u}_s^*, \mathbf{d}_s^*, \mathbf{w}_s^*, \boldsymbol{\theta}_s^*)\}_{s=1}^T$, and use them in the Monte Carlo method as

$$f(y \mid \mathbf{y}) \approx \sum_{s=1}^{T} f(y \mid \mathbf{u}_s^*, \mathbf{d}_s^*, \mathbf{w}_s^*, \boldsymbol{\theta}_s^*)/T,$$

approximating in this way the expected a posteriori (EAP) distribution function for a new observation. It is customary to discard the first state variables to reduce the correlation with the initialization parameters, also to take spaced out observations to reduce serial correlation between variables. For our example this translates to

$$f(y \mid \mathbf{y}) \approx \sum_{s=1}^{n} f(y \mid \mathbf{u}_{t+sl}^*, \mathbf{d}_{t+sl}^*, \mathbf{w}_{t+sl}^*, \boldsymbol{\theta}_{t+sl}^*)/n, \tag{9}$$

where t is the burn-in period of initial steps to be discarded, l is the spacing between iterations and $n = \text{floor}((T-t)/l)$.

We can also approximate the distribution function using the maximum a posteriori (MAP), which can be estimated within the sampled Gibbs steps as

$$f_{MAP}(y \mid \mathbf{y}) \approx f(y \mid \mathbf{u}_s^*, \mathbf{d}_s^*, \mathbf{w}_s^*, \boldsymbol{\theta}_s^*), \tag{10}$$

with

$$s = \operatorname*{argmax}_{t \in \{1,\dots,T\}} f(\mathbf{y}, \mathbf{u}_t^*, \mathbf{d}_t^*, \mathbf{w}_t^*, \boldsymbol{\theta}_t^*).$$

Mean-field variational Bayes

The variational Bayes method is a functional approach to approximate the posterior distribution $f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})$ by means of a variational distribution $g(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w})$. The mean-field framework is a simplification over the search space of variational functions where we assume independence between each parameter. This means that the variational distribution can be factored as the product $g(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w}) = g(\mathbf{d})g(\boldsymbol{\theta})g(\mathbf{w})$. Through this method we look to minimize the following Kullback-Leiber (KL) divergence (Kullback and Leibler 1951)

$$KL(g(\mathbf{d})g(\boldsymbol{\theta})g(\mathbf{w}) \mid\mid f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})) = \mathbb{E}_g[\log g(\mathbf{d})g(\boldsymbol{\theta})g(\mathbf{w}) - \log f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})].$$

Since the KL divergence is not symmetrical, an order for the arguments must be chosen, and since usually $f(\mathbf{d}, \boldsymbol{\theta}, \mathbf{w} \mid \mathbf{y})$ is intractable (otherwise we wouldn't be recurring to approximations), it is easier to calculate the expected values with respect to g and so the stated order is preferred. Minimizing the KL divergence is equivalent to maximizing the evidence lower bound

$$\begin{aligned}\mathcal{L}(g) =& \mathbb{E}_g[\log f(\mathbf{y}, \mathbf{d}, \boldsymbol{\theta}, \mathbf{w}) - \log g(\mathbf{d})g(\boldsymbol{\theta})g(\mathbf{w})] \\ =& \mathbb{E}_g[\log f(\mathbf{y} \mid \mathbf{d}, \boldsymbol{\theta})] \\ &+ \mathbb{E}_g[\log f(\mathbf{d} \mid \mathbf{w})] - \mathbb{E}_g[\log g(\mathbf{d})] \\ &+ \mathbb{E}_g[\log f(\boldsymbol{\theta})] - \mathbb{E}_g[\log g(\boldsymbol{\theta})] \\ &+ \mathbb{E}_g[\log f(\mathbf{w})] - \mathbb{E}_g[\log g(\mathbf{w})].\end{aligned} \tag{11}$$

This maximization can be done using coordinate descent, optimizing each of the marginal distributions $g(\mathbf{d})$, $g(\boldsymbol{\theta})$ and $g(\mathbf{w})$; one at a time, via the following relation for the optimal distribution of g (Bishop 2006, p. 466),

$$\log g(z) = E_g[\log f(\mathbf{y}, \mathbf{d}, \boldsymbol{\theta}, \mathbf{w}) \mid z] + \text{const.},$$

which, after substituting the model variables yields,

$$\begin{aligned} g(\mathbf{d}) &\propto \exp \mathbb{E}_g[\log f(\mathbf{y} \mid \mathbf{d}, \boldsymbol{\theta}) + \log f(\mathbf{d} \mid \mathbf{w})], && (12)\\ g(\boldsymbol{\theta}) &\propto \exp \mathbb{E}_g[\log f(\mathbf{y} \mid \mathbf{d}, \boldsymbol{\theta}) + \log f(\boldsymbol{\theta})], && (13)\\ g(\mathbf{w}) &\propto \exp \mathbb{E}_g[\log f(\mathbf{d} \mid \boldsymbol{\theta}, \mathbf{w}) + \log f(\mathbf{w})]. && (14) \end{aligned}$$

Here we need to make another simplification. Since (12) defines a discrete distribution, all the terms must be computed in order to find the normalizing variable which is not numerically feasible. For this reason, a truncation (k) over the weighting structure is set such that $\mathbb{P}_g[\sum_{j=1}^{k} w_j = 1] = 1$, this is done over the stick breaking representation as $\mathbb{P}_g\left[v_k = 1\right] = 1$, giving us the variational distribution for $\mathbf{d}$,

$$g(\mathbf{d}) = \prod_{i=1}^{N} \prod_{j=1}^{k} r_{i,j}^{\mathbb{1}(d_i = j)},$$

with $r_{i,j} = \rho_{i,j}/(\sum_{l=1}^{k} \rho_{i,l})$ and $\log \rho_{i,j} = \mathbb{E}_g[\log w_j] - \frac{1}{2}\mathbb{E}_g[\log|\Sigma_j|] - \frac{1}{2}\mathbb{E}_g[(y_i - \mu_j)^\top \Sigma_j^{-1}(y_i - \mu_j)]$. This is a finite discrete distribution and taking means over it can be done with ease. Thanks to the conjugacy of the Normal-Inverse Wishart we get that each element of $\boldsymbol{\theta}$ is again a Normal-Inverse Wishart distribution,

$$g(\boldsymbol{\theta}) = \prod_{j=1}^{k} \text{NIW}(\theta_j \mid m_j, \lambda_j, \Psi_j, \nu_j),$$

with parameters

$$\begin{aligned} \lambda_j &= \lambda + n_j,\\ m_j &= \lambda_j^{-1}(\lambda m + n_j \overline{y}_j),\\ \Psi_j &= \Psi_0 + n_j S_j + \lambda_j^{-1} \lambda n_j (\overline{y}_j - m)(\overline{y}_j - m)^\top,\\ \nu_j &= \nu + n_j. \end{aligned}$$

here $n_j = \sum_{i=1}^{N} r_{i,j}$ is the variational mean of elements assigned to the j'th component, $\overline{y}_j = n_j^{-1} \sum_{i=1}^{N} r_{i,j} y_i$ is the variational mean of assigned elements and $S_j = n_j^{-1} \sum_{i=1}^{N} r_{i,j}(y_i - \overline{y}_j)(y_i - \overline{y}_j)^\top$ is the variational variance. An explicit distribution for $\mathbf{w}$ can't be given, but the variational optimization follows the relation

$$g(\mathbf{w}) \propto f(\mathbf{w}) \prod_{j=1}^{k} v_j^{a'_j}(1 - v_j)^{b'_j}, \tag{15}$$

which, as shown in Appendix B, has an analytic expression for most of the BNP models here discussed, so we can refer instead to the parameters of the variational distributions as the optimizing variational parameters. In particular for the Dirichlet process, using the

Algorithm 2 Variational Bayes

```
procedure FIT_VARIATIONAL(y)
    k ← number of groups
    iter ← 0
    ELBO ← $\infty$
    ELBO_change ← $\infty$
    Initialize variational parameters with dimension k
    while ELBO_change $\geq$ tolerance and iter<max_iter do
        iter ← iter+1
        Maximize the variational parameters of $\mathbf{w}$
        Maximize the variational parameters of $\boldsymbol{\theta}$
        Maximize the variational parameters of $\mathbf{d}$
        Compute ELBO
        ELBO_change ← previous_ELBO − ELBO.
    end while
end procedure
```

optimization given by (15) reaches the same variational parameters as those found by Blei and Jordan (2006).

In Algorithm 2 we give the pseudocode of the variational Bayes algorithm used to fit $g(\mathbf{w})$, $g(\boldsymbol{\theta})$ and $g(\mathbf{d})$. Once we have approximated the optimal variational parameters, inference can be done using the variational distribution as if it were the true posterior distribution of each variable. As an example, we can estimate the posterior distribution of a new observation, $f(y \mid \mathbf{y})$, by replacing the posterior distribution

$$\begin{aligned} f(y \mid \mathbf{y}) &= \mathbb{E}[f(y \mid \mathbf{d}, \boldsymbol{\theta}) \mid \mathbf{y}] \\ &\approx \mathbb{E}_g[f(\hat{y} \mid \mathbf{d}, \boldsymbol{\theta})], \end{aligned}$$

and taking that last expected value with respect to the distribution $g(\mathbf{d}, \theta) = g(\mathbf{d})g(\boldsymbol{\theta})$.

3. Software design

The **pyrichlet** package is coded in the Python programming language and uses the packages **numpy** (Harris *et al.* 2020) for numerical computations, **scipy** (Virtanen *et al.* 2020) for some probabilistic functions, and **pandas** (McKinney 2010) `DataFrame` objects as an optional format for reading user data. We divided it in two modules, `weight_models`, and `mixture_models`; the former implements the code to update the conditional weighting structure $f(\mathbf{w} \mid \mathbf{d})$, draw samples from it, and additionally for some models, compute the optimal variational distribution $g(\mathbf{w})$; the later implements a Gibbs sampler with three sampling steps: update $\boldsymbol{\theta}$, update $\mathbf{w}$ and update $\mathbf{d}$. A coordinate descent algorithm is performed to iteratively update the optimal variational distributions for $\boldsymbol{\theta}$, $\mathbf{w}$ and $\mathbf{d}$. This division was intended to gain code modularity and to ease the development of new weighting model classes, from the point of view of both the developer, and the user seeking to implement a weighting model class child. Mixture model classes can be accessed directly from **pyrichlet** without importing `mixture_models`. The `weight_models` module is only meant to be called internally, but is left exposed for the user to explore and extend the package.

Creating a `mixture_models` class object is straightforward, we need to import the package and call the respective initialization,

```
>>> import pyrichlet
>>> import numpy as np
>>> rng = np.random.default_rng(0)
>>> mixture = pyrichlet.DirichletProcessMixture(rng=rng)
```

here we use the variable `rng` for results replication. The different classes found in the `mixture_models` module are wrappers of the class `BaseGaussianMixture` and only differ in the underlying `weight_models` class used, the parameters of a particular model are delegated as named parameters to the mixture model initialization.

From a user point of view, the package main functions are the distinct methods of the `BaseGaussianMixture` class. For doing density estimation these are `gibbs_eap_density` and `gibbs_map_density`, which return the posterior density using Gibbs sampling; and `var_eap_density` and `var_map_density` which return the variational posterior distribution. Meanwhile for clustering, the methods `gibbs_eap_spectral_consensus_cluster`, `gibbs_map_cluster`, `var_eap_spectral_consensus_cluster` and `var_eap_cluster` return a vector of labels. To be able to call any of the previous methods, either `fit_gibbs` or `fit_variational` must be respectively ran before to fit the mixture model over a particular dataset `y`. This sets the internal variable `gibbs_fitted` or `var_fitted` to `True`,

```
>>> y = np.concatenate([rng.normal(1, 1, size=100),
...                     rng.normal(4, 2, size=200)])
>>> y = y.reshape(300, -1)
>>> mixture.fit_gibbs(y, init_groups=2)
>>> mixture.var_fitted

False

>>> mixture.gibbs_fitted

True
```

here the fitting functions use the model hyperparameters defined on initialization, or if none was defined, a default non informative value. After fitting, users can use any of the main functions, for example,

```
>>> density = mixture.gibbs_eap_density([1, 2])
>>> density

array([0.16807104, 0.1917727 ])
```

which returns the EAP density at the locations $\{1, 2\}$.

3.1. Classes

There is a one to one relation between the classes in `weight_models`, and the implementing mixture in `mixture_models` as seen in Table 2 which is an exhaustive list of all the classes in **pyrichlet**.

weight_models	mixture_models
DirichletDistribution	DirichletDistributionMixture
DirichletProcess	DirichletProcessMixture
PitmanYorProcess	PitmanYorMixture
GeometricProcess	GeometricProcessMixture
BetaInBeta*	BetaInBetaMixture
BetaInDirichlet*	BetaInDirichletMixture
BetaBinomial*	BetaBinomialMixture
FrequencyWeighting	FrequencyWeightedMixture
EqualWeighting	EqualWeightedMixture

Table 2: Classes in **pyrichlet**. The models marked with an asterisk do not have a variational implementation

The classes in `weight_models` are implementations of the abstract class `BaseWeight`, and classes in `mixture_models` initialize their respective weighting model class and pass it as an argument to the constructor of the class `BaseGaussianMixture`.

3.2. Weighting models methods

All the weighting model classes are based on the abstract class `BaseWeight`, which has two abstract methods: `random(size)` which returns a random draw of $\mathbf{w}$ up to the `size`'th element using the prior or the posterior distribution (8) of the weighting model (see Appendix A) if `fit` has already been called; and `complete(size)`, which extends the length of $\mathbf{w}$ with observations from the prior. The class method `fit_variational` takes an array of the variational distributions of $\mathbf{d}$ and calculates the optimal variational distribution $\mathbf{w}$ which particular expression can be found in Appendix B for the implementing models. After calling the previous method, the following methods can be used

- `variational_mean_w_j(j)`: Returns $\mathbb{E}_g[w_j]$.
- `variational_mode_w_j(j)`: Returns the mode of $g(w_j)$.
- `variational_mean_log_w_j(j)`: Returns $\mathbb{E}_g[w_j]$.
- `variational_mean_log_p_d__w()`: Returns $\mathbb{E}_g[\log p(\mathbf{d} \mid \mathbf{w})]$.
- `variational_mean_log_p_w()`: Returns $\mathbb{E}_g[\log p(\mathbf{w})]$.
- `variational_mean_log_q_w()`: Returns $\mathbb{E}_g[\log g(\mathbf{w})]$.

The constructor takes the hyperparameters of their weighting model as defined in Section 2.1 or in Appendix C.

3.3. Mixture models methods

The arguments for the initialization of a derived class of `BaseGaussianMixture` are: the hyperparameters of the Normal-Inverse Wishart, the parameters for the Gibbs sampler or variational Bayes methods, a boolean, `show_progress`, to toggle the display of progress, and a random state or seed, `rng`, to be able to replicate a particular result. Explicitly, the arguments needed to initialize a `BaseGaussianMixture` class are:

- `mu_prior`: The centering parameter (m_0) of the prior Normal-inverse Wishart distribution. It must be a float or an array depending on the dimension of the observations, its default value is `None`. If equal to `None`, the sample mean of the observations passed to the fit function will be used.
- `lambda_prior`: The precision parameter (λ_0) of the prior Normal-inverse Wishart distribution. It must be a float, its default value is 1.
- `psi_prior`: The inverse scale matrix (Ψ_0) of the prior Normal-inverse Wishart distribution. It must be a float or a matrix depending on the dimension of the observations, its default value is `None`. If equal to `None`, the sample variance-covariance matrix of the observations passed to the fit function will be used.
- `nu_prior`: The degrees of freedom (ν_0) of the prior normal-inverse Wishart distribution. It must be a float, its default value is `None`. If equal to `None`, the dimension of the scale matrix will be used.
- `total_iter`: The total number of steps to take in the Gibbs sampler algorithm or the maximum number of steps to take in the coordinate descent algorithm. It must be an integer, its default value is 1000.
- `burn_in`: The number of initial steps to discard in the Gibbs sampler before starting to save the state parameters. It must be an integer, its default value is 100.
- `subsample_steps`: The number of steps to take in the Gibbs sampler before saving the state parameters. It must be an integer, its default value is 1.
- `show_progress`: A flag to say whether to display the steps of the fitting method's loop or not. It must be a boolean, its default value is `False`.
- `rng`: The random state of the mixture, analogous to setting a seed in a pseudo-random number generator algorithm. It must be an integer, its default value is `None`. If equal to `None`, no seed is set.

Additionally, in any `BaseGaussianMixture` derived class, the parameters of its particular base weighting model are needed.

The method `fit_gibbs` takes an array of observations as its input and runs a Gibbs sampler as described in Algorithm 1. The optional arguments for this method are:

- `init_groups`: The maximum group label to assign to $\mathbf{d}$ in its initialization. The default is to take the number of observations to fit as the maximum.
- `warm_start`: If the method is called a second time, this flag states whether to continue the sampling process from a past run and assume that all the variables are already initialized or to do a fresh start and forget any previously saved states.

- `show_progress`: This flag serves the same purpose as its initialization counterpart and is repeated here as an override.
- `init_method`: This is a string to choose the variable initialization method for $\mathbf{d}$ and is one of:
  - `"random"`: The initialization is taken at random from the prior distribution.
  - `"kmeans"`: Runs a k-means clustering to define $\mathbf{d}$.
  - `"variational"`: Fits a variational distribution and uses the MAP clustering as the initialization.

The method `fit_variational` also takes an array of observations as its input and runs a coordinate descent algorithm to optimize the variational distributions as described in Algorithm 2. It's optional arguments are:

- `n_groups`: The variational truncation parameter, k.
- `warm_start`: For subsequent calls, whether to take the previously fitted variational distributions as the initialization, or to do a fresh start.
- `show_progress`: Same as for `fit_gibbs`.
- `tol`: The minimum change in ELBO between iterations needed to assume that the optimization method has converged.
- `method`: A string to choose the variational initialization for $\mathbf{d}$ and one of:
  - `"kmeans"`: Runs a k-means clustering to define the initial distribution of $\mathbf{d}$ as a constant variable.
  - `"random"`: Sets a discrete uniform distribution as the variational initialization.

After fitting using the Gibbs sampler, the user can access the class variable `sim_params` which contains all the saved state variables as an array, to perform a Monte Carlo approximation over a quantity of interest to get the EAP or MAP estimates as showed in (9) and (10); or use the class variable `map_sim_params`, which is a dictionary with the set of parameters that reached the highest model likelihood within the different sampled states. The analogous class variables for the variational method are: `var_d`, which is an array of the discrete variational probability distributions of $\mathbf{d}$; `var_theta`, an array of parameters for a Normal-inverse Wishart distribution corresponding to the fitted variational distribution of $\boldsymbol{\theta}$; and the variational methods of the weighting model as described in Section 3.2.

Alternatively, the following methods can be used to draw some insights from the appropriate fitted model over an array of locations `y`:

- `gibbs_eap_density`: Takes an array of locations and, optionally the number of state parameters to use, and returns the EAP density estimation at those points.
- `gibbs_map_density`: Takes an array of locations and returns the MAP density estimation at those points.

- gibbs_eap_affinity_matrix: Takes an array of locations and returns the mean posterior affinity matrix using the saved state variables $\mathbf{d}^*$.
- gibbs_eap_spectral_consensus_cluster: Takes an array of locations and a number of clusters and uses their EAP affinity matrix to perform spectral clustering returning the resulting assignations.
- gibbs_map_cluster: Takes an array of locations and uses the MAP parameters to estimate their clustering. Optionally a boolean full can be passed to return a tuple with the clusters and assignation uncertainties.
- var_eap_density: Takes an array of locations and calculates their variational mean density.
- var_map_density: Takes an array of locations and computes their variational mode density.
- var_eap_affinity_matrix: Takes an array of locations and returns the posterior affinity matrix using the variational probability of group matches.
- var_eap_spectral_consensus_cluster: Takes an array of locations and a number of clusters and uses their variational EAP affinity matrix to perform spectral clustering returning the resulting assignations.
- var_map_cluster: Takes an array of locations and returns their variational mode assignations.

3.4. Extensibility

The user can extend the package to a new mixing model class with an arbitrary weighting model $f(\mathbf{w})$, and use a Gibbs sampler to fit its posterior by implementing a derived class of BaseWeight and passing it to the constructor of BaseGaussianMixture. This can be done by deriving the conditional distribution $f(\mathbf{w} \mid \mathbf{d})$ and use it to implement the methods __init__, random and complete. To be able to fit the model using variational Bayes, all the variational methods need to be implemented.

As an example, to create a basic mixture model with a Dirichlet distribution weighting model (differing with the implemented model by the lack of variational inference methods), the following code can be used to declare the weighting model,

```
>>> from pyrichlet import BaseWeight
>>> class BasicDirichletDistribution(BaseWeight):
...   def __init__(self, n=1, alpha=1, rng=None):
...     super().__init__(rng=rng)
...     self.n = n
...     self.alpha = np.array([alpha] * self.n, dtype=np.float64)
...
...   def random(self, size=None):
...     if len(self.d) > 0:
...       a_c = np.bincount(self.d)
...       a_c.resize(len(self.alpha), refcheck=False)
...       self.w = self._rng.dirichlet(self.alpha + a_c)
...     else:
...       self.w = self._rng.dirichlet(self.alpha)
...     return self.w
...
...   def complete(self, size=None):
...     super().complete(size)
...     if len(self.w) == 0:
...       self.random()
...     return self.w
```

and then use it to define a mixture model class,

```
>>> from pyrichlet.mixture_models._base import BaseGaussianMixture
>>> class BasicDirichletDistributionMixture(BaseGaussianMixture):
>>>   def __init__(self, *, n=1, alpha=1, rng=None, **kwargs):
...     weight_model = BasicDirichletDistribution(n=n, alpha=alpha,
...                                               rng=rng)
...     self.n = n
...     super().__init__(weight_model=weight_model, rng=rng,
...                      **kwargs)
```

After defining `BasicDirichletDistributionMixture`, this new class can be used as any other weighting model:

```
>>> mixture = BasicDirichletDistributionMixture(n=5)
```

This particular custom class can be fitted using Gibbs sampling via `fit_gibbs`, but will throw a `NotImplementedError` if we try to fit its variational distribution.

4. Usage

To illustrate how to use **pyrichlet**, we will use the `palmerpenguins` dataset described in Horst, Hill, and Gorman (2022) and originally published in Gorman, Williams, and Fraser (2014), consisting on data of 344 male and female *Pygoscelis* penguins from three islands in the Palmer Archipelago, Antarctica, 342 of which have bill, flipper and body mass measurements.

We will start by fitting a Beta in Beta process mixture using a Gibbs sampler, then estimate the MAP density in the data bounding region, cluster the observed data points, and finally produce a scatter plot colored by group and with an overlay of each component's density contour.

Firstly we need to import our package, **pyrichlet**, as well as **numpy** for result repeatability by creating a random generator with a set seed. The helper function `load_penguins` is used for convenience to load the data.

```
>>> import pyrichlet
>>> import numpy as np
>>> y = pyrichlet.utils.load_penguins()[0]
```

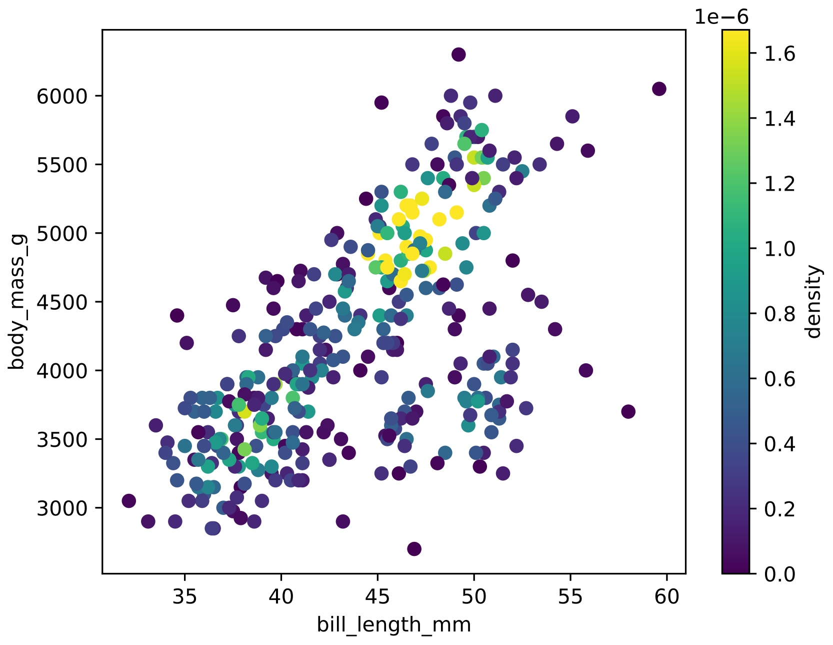

Figure 1: Scatter plot of the `palmerpenguins` dataset colored by MAP estimated density.

then we initialize the chosen model (`BetaInBetaMixture` in this case) with its hyperparameters and pass our database to the fitting function,

```
>>> mixture = pyrichlet.BetaInBetaMixture(x=0.5, rng=0)
>>> mixture.fit_gibbs(y, init_groups=3)
```

After having fitted the model we can now do density estimation and clustering.

4.1. Density estimation

If we call the method `gibbs_map_density()` without arguments, we will get the estimation at each location of the database, otherwise we would pass a list of 4-dimensional arrays at which the density is to be estimated. After getting the MAP density, we can visualize it using a scatter plot from the **matplotlib** package (Hunter 2007) for the first and last variables, `bill_length_mm` and `body_mass_g`, and color each point with its corresponding density as shown in Figure 1.

```
>>> import matplotlib.pyplot as plt
>>> density = mixture.gibbs_map_density()
>>> plt.scatter(y.iloc[:, 0], y.iloc[:, 3], c=density)
>>> cbar = plt.colorbar()
>>> plt.clim(0, density.max() / 1.5)
>>> cbar.set_label('density')
>>> plt.xlabel(y.columns[0])
>>> plt.ylabel(y.columns[3])
>>> plt.show()
```

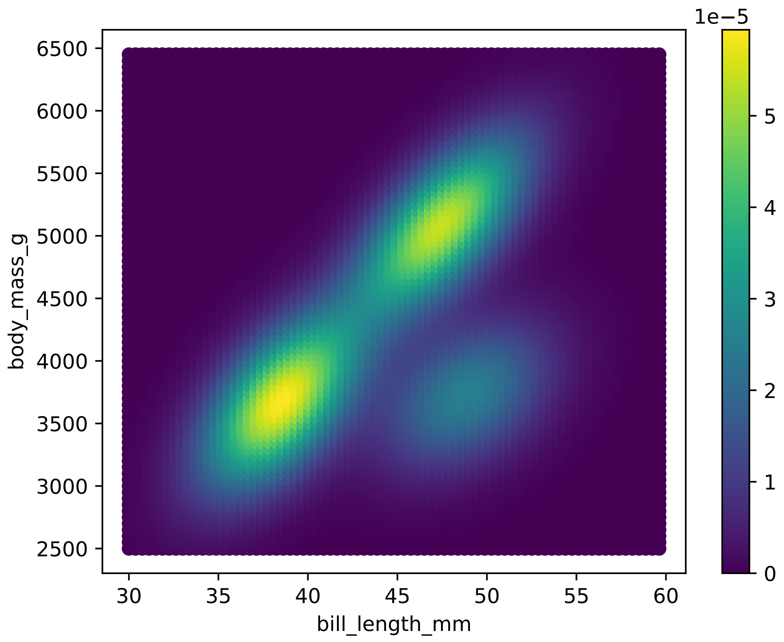

Figure 2: Scatter plot of the projected, Gibbs fitted MAP density for the variables `bill_length_mm` and `body_mass_g`.

Using the class variable `map_sim_params` we can calculate the MAP projected bivariate density for `bill_length_mm` and `body_mass_g`, which is a bivariate normal mixture with the original weighting structure but projected components, as shown in Figure 2, where we can see that, for this projection, there is an overlapping region between the lower left and upper right components,

```
>>> from scipy.stats import multivariate_normal
>>> XY = np.mgrid[30:60:0.375, 2500:6500:50]
>>> y_space = XY.reshape(2, -1).T
>>> w = mixture.map_sim_params['w']
>>> theta = mixture.map_sim_params['theta']
>>> density = []
>>> for j in range(len(w)):
...   density.append(
...     multivariate_normal.pdf(y_space,
...                                 theta[j][0][[0, 3]],
...                                 theta[j][1][:, [0, 3]][[0, 3], :],
...                                 1))
>>> density = w @ density
>>> plt.scatter(y_space[:, 0], y_space[:, 1], c=density)
>>> cbar = plt.colorbar()
>>> plt.xlabel(y.columns[0])
>>> plt.ylabel(y.columns[3])
>>> plt.show()
```

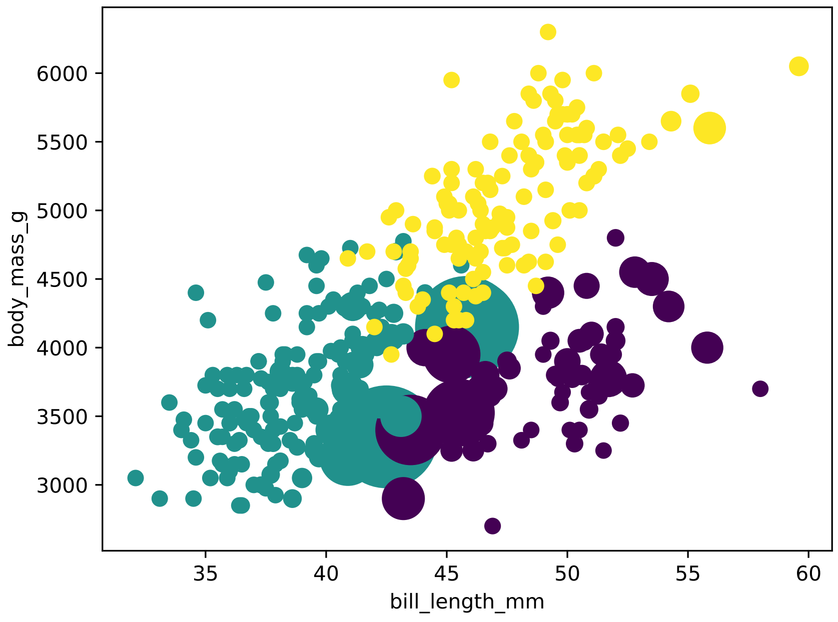

Figure 3: Scatter plot of the `palmerpenguins` dataset colored by MAP clustering and sized by assignation uncertainty.

4.2. Clustering

We can use the previously fitted object `mixture` to get the MAP clustering and its respective assignation uncertainties, $\mathbb{P}(d_i \neq j \mid \mathbf{w}_s^*, \boldsymbol{\theta}_s^*)$, which serves to measure the probability of having an assignation different to the one given to the i-th observation. We plot a scatter colored by the respective clustering with size based on the uncertainty in Figure 3. The biggest points are those where there is not as much confidence in its assignation as with other points, and we can see that the overlapping density discussed previously does not pose a challenge when trying to assign the observed points, but a seemingly separate region between the lower components does.

```
>>> group, uncertainty = mixture.gibbs_map_cluster(y, full=True)
>>> order = np.argsort(-uncertainty)
>>> plt.scatter(y.iloc[:, 0], y.iloc[:, 3], c=group,
...             s=5000 * (0.01 + uncertainty))
>>> plt.xlabel(y.columns[0])
>>> plt.ylabel(y.columns[3])
>>> plt.show()
```

By calling the plotting method `gibbs_map_pairplot` we can get a summary as a matrix of plots, with the diagonal filled with plots of the corresponding marginalized density as well as its decomposition in Gaussian components; and the off diagonal with the pairwise projections consisting of a scatter plot colored by cluster and a 95% bivariate Gaussian level curve for each displayed component. This is shown in Figure 4.

```
>>> mixture.gibbs_map_pairplot()
```

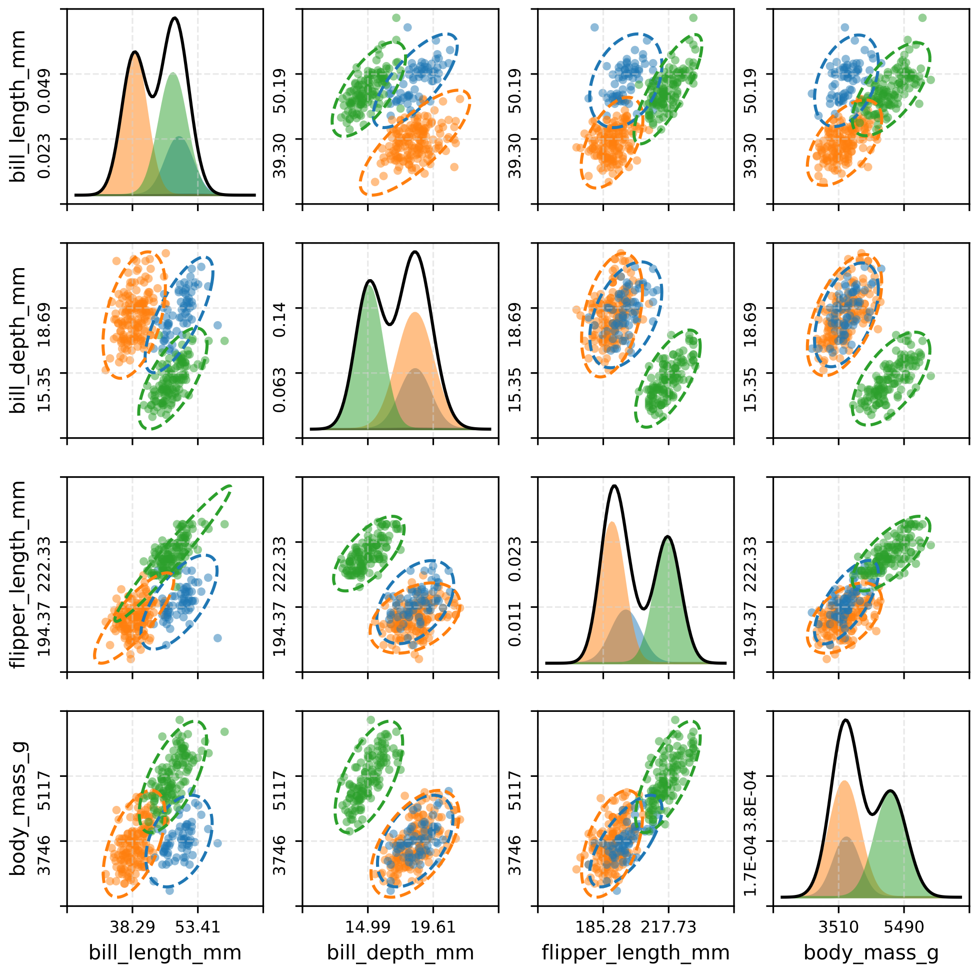

Figure 4: Pair plot of the `palmerpenguins` dataset using the Gibbs fitted MAP clustering and projected densities.

```
>>> plt.show()
```

It is important to emphasize that the estimated density shown at each diagonal plot is the marginalization of each component in the estimated MAP distribution given by Equation 10 and scaled by their respective weighting.

4.3. Package comparison and running times

To see how **pyrichlet** compares to other packages, we have computed the running times and clustering results for the `palmerpenguins` dataset as a benchmark.

All the weighting models of **pyrichlet** have been included in the comparison as well as most of the packages showed in Table 1, with the exception of the R package **DPpackage** and the Python package **Mixture-Models** due to technical difficulties on their installation, while the package **BayesMix** was run only through its Python wrapper. This comparison uses the default specifications of each library. The running times using the Gibbs sampling fitting method are

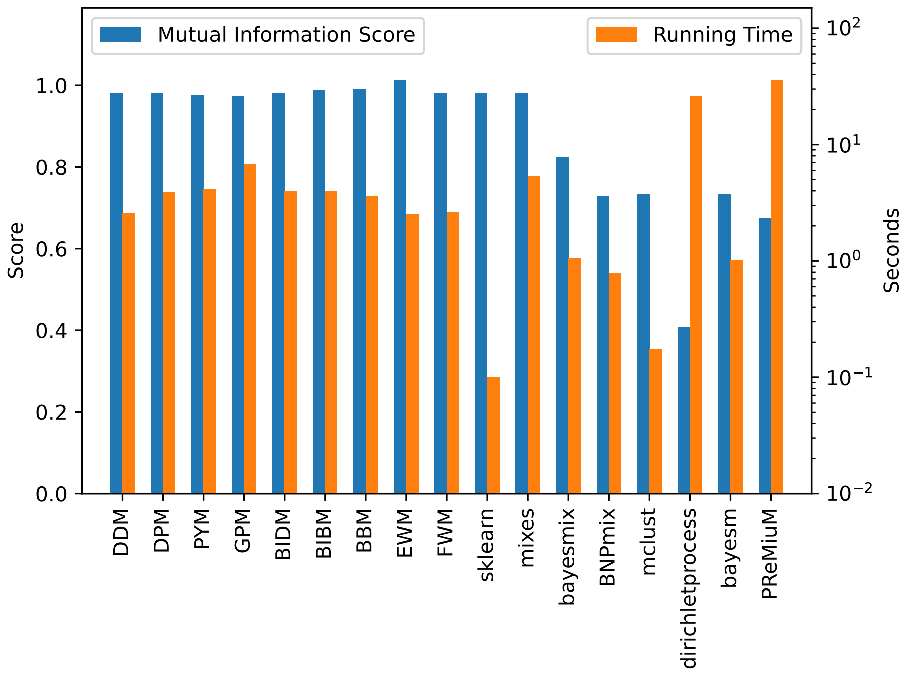

Figure 5: Mutual information score and running times of all weighting models implemented in **pyrichlet** and several other packages, fitted against the `palmerpenguins` dataset. The nine leftmost methods are the ones included in **pyrichlet** and correspond to the different weighting structures fitted via Gibbs sampling.

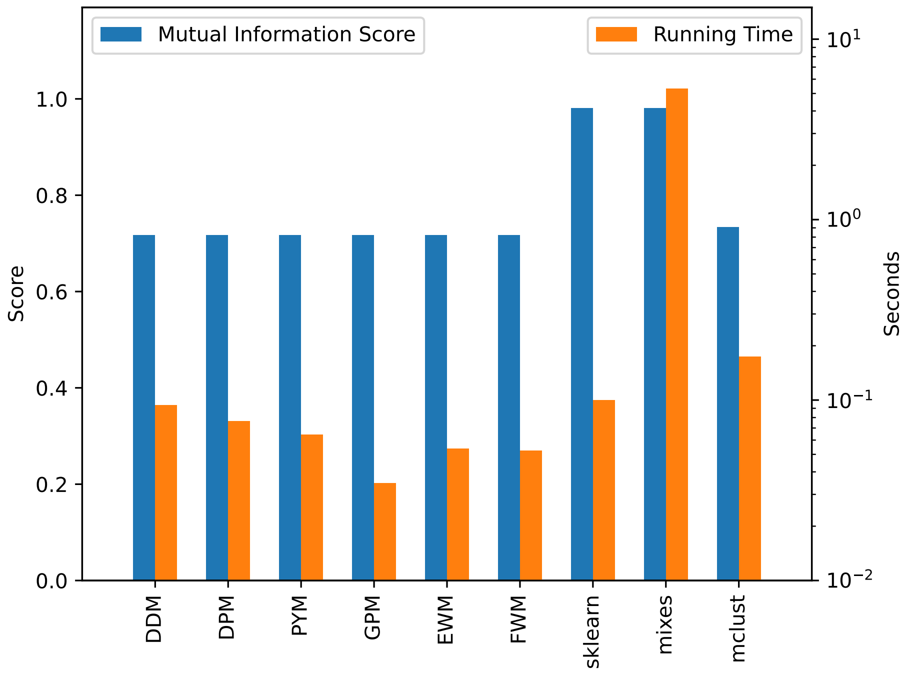

Figure 6: Mutual information score and running times of a subset of the weighting models implemented in **pyrichlet** and several other packages that also implement the variational fitting method, fitted against the `palmerpenguins` dataset. The six leftmost methods are the ones included in **pyrichlet** that implement a variational fitting.

shown in Figure 5 together with the mutual information score (see Shannon 1948) against the classification given in the `palmerpenguins` dataset. For the weighting models that have a variational fitting implementation, their running times and mutual information score can be seen in Figure 6. The acronyms used in Figure 5 and Figure 6 stand for Dirichlet distribution mixture (DDM), Dirichlet process mixutre (DPM), Pitman-Yor mixture (PYM), geometric process mixture (GPM), beta in Dirichlet mixture (BIDM), beta in beta mixture (BIBM), beta binomial mixture (BBM), equal weighted mixture (EWM) and frequency weighted mixture (FWM).

5. Final remarks

We have introduced the Python package **pyrichlet** which introduces several Bayesian non parametric models, some of which introduced in recent years. Part of the strengths of this software is the ability to fit arbitrary database densities with greater flexibility compared to more traditional methods like kernel density estimation. The memory usage scalability of the implemented fitting methods is proportional to that of the k-means algorithm with respect to the database size, but a major pitfall (inherent to model based approaches) is the impact of the curse of dimensionality, particularly in the Gibbs sampling method, where the number of steps must be adjusted proportionally.

Computational details

The results shown in this paper were obtained using Python 3.10.9 with the **pyrichlet** 0.0.9 package on an Intel i5-9300H CPU @ 2.40GHz processor. The **pyrichlet** package is available in the Python Package Index (PyPI) at `https://pypi.org/project/pyrichlet/`.

Acknowledgments

The authors gratefully acknowledge the support of PAPIIT grants IN100823 and IA101124. The first author was also supported by a CONAHCYT PhD scholarship.

References

Beraha M, Guindani B, Gianella M, Guglielmi A (2023a). ***BayesMixPy**: A Python Interface to **BayesMix***. URL `https://github.com/bayesmix-dev/bayesmix/tree/v0.2.0/python`.

Beraha M, Guindani B, Gianella M, Guglielmi A (2023b). ***BayesMixR**: An R Interface to **BayesMix***. URL `https://github.com/bayesmix-dev/bayesmix/tree/v0.2.0/R`.

Beraha M, Guindani B, Gianella M, Guglielmi A (2025). "**BayesMix**: Bayesian Mixture Models in C++." *Journal of Statistical Software*, **112**(9), 1–41. `doi:10.18637/jss.v112.i09`.

Bishop CM (2006). *Pattern Recognition and Machine Learning*. Springer-Verlag.

Bissiri PG, Ongaro A (2014). "On the Topological Support of Species Sampling Priors." *Electronic Journal of Statistics*, **8**(1). doi:10.1214/14-ejs912.

Blackwell D, MacQueen JB (1973). "Ferguson Distributions via Polya Urn Schemes." *The Annals of Statistics*, **1**(2), 353–355. doi:10.1214/aos/1176342372.

Blei DM, Jordan MI (2006). "Variational Inference for Dirichlet Process Mixtures." *Bayesian Analysis*, **1**(1), 121–143. doi:10.1214/06-ba104.

Canale A, Corradin R, Nipoti B (2022). "Importance Conditional Sampling for Pitman-Yor Mixtures." *Statistics and Computing*, **32**(3), 40. doi:10.1007/s11222-022-10096-0.

Corradin R, Canale A, Nipoti B (2021). "**BNPmix**: An R Package for Bayesian Nonparametric Modeling via Pitman-Yor Mixtures." *Journal of Statistical Software*, **100**(15), 1–33. doi:10.18637/jss.v100.i15.

Escobar MD, West M (1995). "Bayesian Density Estimation and Inference Using Mixtures." *Journal of the American Statistical Association*, **90**(430), 577–588. doi:10.1080/01621459.1995.10476550.

Ferguson TS (1973). "A Bayesian Analysis of Some Nonparametric Problems." *The Annals of Statistics*, **1**(2). doi:10.1214/aos/1176342360.

Ferguson TS (1983). "Bayesian Density Estimation by Mixtures of Normal Distributions." In *Recent Advances in Statistics*, pp. 287–302. Elsevier. doi:10.1016/b978-0-12-589320-6.50018-6.

Fuentes-García R, Mena RH, Walker SG (2010). "A New Bayesian Nonparametric Mixture Model." *Communications in Statistics – Simulation and Computation*, **39**(4), 669–682. doi:10.1080/03610910903580963.

Gelfand AE, Kottas A (2002). "A Computational Approach for Full Nonparametric Bayesian Inference under Dirichlet Process Mixture Models." *Journal of Computational and Graphical Statistics*, **11**(2), 289–305. doi:10.1198/106186002760180518.

Gil-Leyva MF, Mena RH (2021). "Stick-Breaking Processes with Exchangeable Length Variables." *Journal of the American Statistical Association*, **0**(0), 1–14. doi:10.1080/01621459.2021.1941054.

Gil–Leyva MF, Mena RH, Nicoleris T (2020). "Beta-Binomial Stick-Breaking Non-Parametric Prior." *Electronic Journal of Statistics*, **14**(1). doi:10.1214/20-ejs1694.

Gorman KB, Williams TD, Fraser WR (2014). "Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis)." *PLOS One*, **9**(3), e90081. doi:10.1371/journal.pone.0090081.

Harris CR, Millman KJ, Van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del Río JF, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE (2020). "Array Programming with **NumPy**." *Nature*, **585**(7825), 357–362. doi:10.1038/s41586-020-2649-2.

Hjort N, Holmes C, Müller P, Walker SG (2010). *Bayesian Nonparametrics*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press. doi:10.1017/cbo9780511802478.

Horst AM, Hill AP, Gorman KB (2022). ***palmerpenguins**: Palmer Archipelago (Antarctica) Penguin Data*. doi:10.32614/CRAN.package.palmerpenguins. R package version 0.1.1.

Hunter JD (2007). "**matplotlib**: A 2D Graphics Environment." *Computing in Science & Engineering*, **9**(3), 90–95. doi:10.1109/mcse.2007.55.

Ishwaran H, James LF (2000). "Approximate Dirichlet Process Computing in Finite Normal Mixtures: Smoothing and Prior Information." *Journal of Computational and Graphical Statistics*, **11**, 508–532. doi:10.1198/106186002411.

Ishwaran H, James LF (2001). "Gibbs Sampling Methods for Stick-Breaking Priors." *Journal of the American Statistical Association*, **96**(453), 161–173. doi:10.1198/016214501750332758.

Jara A, Hanson T, Quintana F, Mueller P, Rosner G (2011). ***DPpackage**: Bayesian Semi-And Nonparametric Modeling in R*. doi:10.32614/CRAN.package.DPpackage. R package version 1.1-7.

Jordan MI, Ghahramani Z, Jaakkola TS, Saul LK (1999). "An Introduction to Variational Methods for Graphical Models." *Machine Learning*, **37**(2), 183–233. doi:10.1023/a:1007665907178.

Kalli M, Griffin JE, Walker SG (2011). "Slice Sampling Mixture Models." *Statistics and Computing*, **21**(1), 93–105. doi:10.1007/s11222-009-9150-y.

Kasa SR (2022). ***Mixture-Models**: A Python Library for Fitting Mixture Models Using Gradient Based Inference*. Python package version 0.0.7, URL https://pypi.org/project/Mixture-Models.

Kasa SR, Rajan V (2020). "Model-Based Clustering Using Automatic Differentiation: Confronting Misspecification and High-Dimensional Data." *arXiv 2007.12786*, arXiv.org E-Print Archive. doi:10.48550/arXiv.2007.12786.

Kullback S, Leibler RA (1951). "On Information and Sufficiency." *The Annals of Mathematical Statistics*, **22**(1), 79–86. doi:10.1214/aoms/1177729694.

Liverani S, Hastie DI, Azizi L, Papathomas M, Richardson S (2015). "**PReMiuM**: An R Package for Profile Regression Mixture Models Using Dirichlet Processes." *Journal of Statistical Software*, **64**(7), 1–30. doi:10.18637/jss.v064.i07.

Lo AY (1984). "On a Class of Bayesian Nonparametric Estimates: I. Density Estimates." *The Annals of Statistics*, **12**(1), 351–357. doi:10.1214/aos/1176346412.

McKenzie P, Alder M (1994). "Selecting the Optimal Number of Components for a Gaussian Mixture Model." In *Proceedings of 1994 IEEE International Symposium on Information Theory*, pp. 393–. doi:10.1109/isit.1994.394626.

McKinney W (2010). "Data Structures for Statistical Computing in Python." In S Van der Walt, J Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56–61. doi:10.25080/majora-92bf1922-00a.

Miller JW, Harrison MT (2014). "Inconsistency of Pitman-Yor Process Mixtures for the Number of Components." *Journal of Machine Learning Research*, **15**(96), 3333–3370. doi:10.1214/14-ba863.

Neal RM (2000). "Markov Chain Sampling Methods for Dirichlet Process Mixture Models." *Journal of Computational and Graphical Statistics*, **9**(2), 249–265. doi:10.1080/10618600.2000.10474879.

Opper M, Saad D (2001). *Advanced Mean Field Methods: Theory and Practice.* Neural Information Processing Series. MIT Press. ISBN 9780262150545.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "**scikit-Learn**: Machine Learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830. doi:10.5555/1953048.2078195.

Pernkopf F, Bouchaffra D (2005). "Genetic-Based EM Algorithm for Learning Gaussian Mixture Models." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **27**, 1344–1348. doi:10.1109/tpami.2005.162.

Pitman J (1996). "Some Developments of the Blackwell-Macqueen URN Scheme." *Lecture Notes-Monograph Series*, **30**, 245–267. doi:10.1214/lnms/1215453576.

Pitman J (2006). *Combinatorial Stochastic Processes*, volume 1875 of *École d'Été de Probabilités de Saint-Flour.* Springer-Verlag, New York. doi:10.1007/b11601500.

Pitman J, Yor M (1997). "The Two-Parameter Poisson-Dirichlet Distribution Derived from a Stable Subordinator." *The Annals of Probability*, **25**(2), 855–900. doi:10.1214/aop/1024404422.

Regazzini E, Lijoi A, Prünster I (2003). "Distributional Results for Means of Normalized Random Measures with Independent Increments." *The Annals of Statistics*, **31**(2), 560–585. doi:10.1214/aos/1051027881.

Ross GJ, Markwick D (2020). ***dirichletprocess**: Build Dirichlet Process Objects for Bayesian Modelling.* doi:10.32614/CRAN.package.dirichletprocess. R package version 0.4.0.

Rossi P (2019). ***bayesm**: Bayesian Inference for Marketing/Micro-Econometrics.* doi:10.32614/CRAN.package.bayesm. R package version 3.1-4.

Scrucca L, Fop M, Murphy TB, Raftery AE (2016). "**mclust** 5: Clustering, Classification and Density Estimation Using Gaussian Finite Mixture Models." *The R Journal*, **8**(1), 289–317. doi:10.32614/rj-2016-021.

Selva F (2020). *Dirichlet Geometric Process.* Master's thesis, UNAM.

Sethuraman J (1994). "A Constructive Definition of Dirichlet Priors." *Statistica Sinica*, **4**(2), 639–650. doi:10.21236/ada238689.

Shannon CE (1948). “A Mathematical Theory of Communication.” *Bell System Technical Journal*, **27**, 623–656. doi:10.1145/584091.584093.

Vila JP, Schniter P (2013). “Expectation-Maximization Gaussian-Mixture Approximate Message Passing.” *IEEE Transactions on Signal Processing*, **61**, 4658–4672. doi:10.1109/tsp.2013.2272287.

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat İ, Feng Y, Moore EW, VanderPlas J, Laxalde D, Perktold J, Cimrman R, Henriksen I, Quintero EA, Harris CR, Archibald AM, Ribeiro AH, Pedregosa F, van Mulbregt P, SciPy 10 Contributors (2020). “**SciPy** 1.0: Fundamental Algorithms for Scientific Computing in Python.” *Nature Methods*, **17**, 261–272. doi:10.1038/s41592-019-0686-2.

Walker SG (2007). “Sampling the Dirichlet Mixture Model with Slices.” *Communications in Statistics – Simulation and Computation*, **36**(1), 45–54. doi:10.1080/03610910601096262.

Xu L, Jordan MI (1996). “On Convergence Properties of the EM Algorithm for Gaussian Mixtures.” *Neural Computation*, **8**(1), 129–151. doi:10.1162/neco.1996.8.1.129.

A. Weighting structures' conditional distributions

Dirichlet distribution

For the Dirichlet distribution, the full conditional distribution of $\mathbf{w}$ without using slices is

$$\begin{aligned} f(\mathbf{w} \mid \ldots) &\propto \text{Dirichlet}(\mathbf{w} \mid \boldsymbol{\alpha}) \prod_{j=1}^{k} w_j^{a_j} \\ &\propto \text{Dirichlet}(\mathbf{w} \mid \boldsymbol{\alpha}^*), \end{aligned}$$

with $\boldsymbol{\alpha}^* = \{\alpha_1 + a_1, \ldots, \alpha_n + a_k\}$.

Dirichlet process

The **u**-collapsed conditional distribution of **v** for the Dirichlet process can be written as

$$\begin{aligned} f(\mathbf{v} \mid \ldots \setminus \mathbf{u}) &\propto \prod_{j\geq 1} \text{Beta}(v_j \mid 1, \alpha) v_j^{a_j} (1 - v_j)^{b_j} \\ &\propto \prod_{j\geq 1} \text{Beta}(v_j \mid 1 + a_j, \alpha + b_j). \end{aligned}$$

Pitman-Yor process

In a Pitman-Yor process, the **u**-collapsed conditional distribution of **v** is given by

$$\begin{aligned} f(\mathbf{v} \mid \ldots \setminus \mathbf{u}) &\propto \prod_{j\geq 1} \text{Beta}(v_j \mid 1 - d, \alpha + jd) v_j^{a_j} (1 - v_j)^{b_j} \\ &\propto \prod_{j\geq 1} \text{Beta}(v_j \mid 1 - d + a_j, \alpha + jd + b_j). \end{aligned}$$

Geometric process

A geometric process can be updated with the **u**-collapsed conditional distribution of p,

$$\begin{aligned} f(p \mid \ldots \setminus \mathbf{u}) &\propto \text{Beta}(p \mid a, b) \prod_{i=1}^{N} w_{d_i} \\ &= \text{Beta}(p \mid a, b) \prod_{i=1}^{N} p(1-p)^{d_i - 1} \\ &\propto \text{Beta}(p \mid a + N, b + \textstyle\sum_{i=1}^{N}(d_i - 1)). \end{aligned}$$

Beta in Dirichlet process

Regarding the sticks of a beta in Dirichlet process, they can be updated with the full conditional distribution for v_j, which holds

$$f(v_j \mid v_{-j}, \ldots \setminus \mathbf{u}) \propto f(v_j \mid v_{-j}) 1(c_j < v_j < c'_j),$$

which corresponds to the truncation of the posterior distribution of a Dirichlet Process after observing v_{-j}.

Beta in beta process

For the beta in beta process, the **u** collapsed conditional distribution of **v** is

$$\begin{aligned} f(\mathbf{v} \mid \ldots \setminus \mathbf{u}) &\propto \prod_{j\geq 1} \text{Beta}\left(v_j \,\middle|\, 1 + p\frac{x}{1-x}, \alpha + (1-p)\frac{x}{1-x}\right) v_j^{a_j}(1-v_j)^{b_j} \\ &\propto \prod_{j\geq 1} \text{Beta}\left(v_j \,\middle|\, 1 + p\frac{x}{1-x} + a_j, \alpha + (1-p)\frac{x}{1-x} + b_j\right), \end{aligned}$$

and the conditional distribution of p collapsed for all v_j with $j > \max d_i$ follows the proportionality

$$f(p \mid v_1, \ldots, v_{\max d_i}) \propto \text{Beta}\left(p \,|a, b\right) \prod_{j=1}^{\max d_i} \text{Beta}\left(v_j \,\middle|\, 1 + p\frac{x}{1-x}, \alpha + (1-p)\frac{x}{1-x}\right),$$

which is not a closed distribution but can be sampled numerically, for example via inverse sampling.

Beta binomial process

The beta Binomial process has a **u** collapsed joint conditional distribution for the variables $(\mathbf{v}, \{B_j\}_{j\geq 1})$ given by

$$\begin{aligned} f(\mathbf{v}, \{B_j\}_{j\geq 1} \mid \ldots \setminus \mathbf{u}) \propto & \text{Beta}(v_0 \mid 1, \alpha) \\ & \times \prod_{j\geq 1} \text{Binomial}(B_j \mid n, v_{j-1}) \\ & \times \text{Beta}(v_j \mid 1 + B_j, \alpha + n - B_j) v_j^{a_j}(1-v_j)^{b_j}. \end{aligned}$$

By marginalizing it can be seen that the **u** collapsed conditional distribution of **v** holds

$$\begin{aligned} f(\mathbf{v} \mid \ldots \setminus \mathbf{u}) \propto & \text{Beta}(v_0 \mid 1, \alpha) v_0^{B_1}(1-v_0)^{n-B_1} \\ & \times \prod_{j\geq 1} \text{Beta}(v_j \mid 1 + B_j, \alpha + n - B_j) v_j^{a_j + B_{j+1}}(1-v_j)^{b_j + n - B_{j+1}} \\ \propto & \text{Beta}(v_0 \mid 1 + B_1, \alpha + n - B_1) \\ & \times \prod_{j\geq 1} \text{Beta}(v_j \mid 1 + B_j + B_{j+1}, \alpha + 2n - B_j - B_{j+1}). \end{aligned}$$

Finally, by removing the proportionally constant terms to the joint distribution, the full conditional distribution for B_j is obtained,

$$f(B_j \mid \ldots) \propto \text{Beta}(v_j \mid 1 + B_j, \alpha + n - B_j)\text{Binomial}(B_j \mid n, v_{j-1}),$$

which is a discrete finite distribution.

B. Weighting structures' variational distributions

Dirichlet distribution

In a Dirichlet distribution, the optimal variational distribution for **w** holds

$$\begin{aligned} g(\mathbf{w}) &\propto \text{Dirichlet}(\mathbf{w} \mid \boldsymbol{\alpha}) \prod_{j=1}^{k} w_j^{a'_j} \\ &\propto \text{Dirichlet}(\mathbf{w} \mid \boldsymbol{\alpha}^*), \end{aligned}$$

with $\boldsymbol{\alpha}^* = \{\alpha_1 + a'_1, \dots, \alpha_n + a'_k\}$.

Dirichlet process

A Dirichlet process has an optimal variational distribution for **v** given by

$$\begin{aligned} g(\mathbf{v}) &\propto \prod_{j=1}^{k} \text{Beta}(v_j \mid 1, \alpha) v_j^{a'_j} (1 - v_j)^{b'_j} \\ &\propto \prod_{j=1}^{k} \text{Beta}(v_j \mid 1 + a'_j, \alpha + b'_j). \end{aligned}$$

Pitman-Yor process

For the Pitman-Yor process, the optimal variational distribution for **v** is

$$\begin{aligned} g(\mathbf{v}) &\propto \prod_{j=1}^{k} \text{Beta}(v_j \mid 1 - d, \alpha + jd) v_j^{a'_j} (1 - v_j)^{b'_j} \\ &\propto \prod_{j=1}^{k} \text{Beta}(v_j \mid 1 - d + a'_j, \alpha + jd + b'_j). \end{aligned}$$

Geometric process

For the Geometric process, the optimal variational distribution for p is

$$\begin{aligned} g(p) &\propto \text{Beta}(p \mid a, b) \prod_{j=1}^{k} w_j^{a'_j} \\ &= \text{Beta}(p \mid a, b) \prod_{j=1}^{k} p^{a'_j} (1 - p)^{(j-1)a'_j} \\ &\propto \text{Beta}\left(p \,\middle|\, a + \textstyle\sum_{i=1}^{N} a'_j, b + \sum_{i=1}^{N} a'_j (j - 1)\right) \\ &= \text{Beta}\left(p \,\middle|\, a + N, b + \textstyle\sum_{i=1}^{N} a'_j (j - 1)\right). \end{aligned}$$

C. Platter representations

Dirichlet distribution

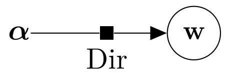

Figure 7: Dirichlet distribution weighting structure plate representation.

Dirichlet process

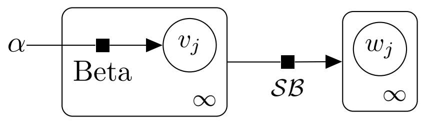

Figure 8: Dirichlet process weighting structure plate representation.

Pitman-Yor process

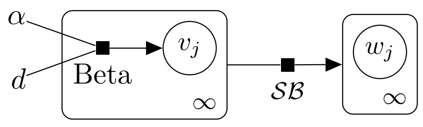

Figure 9: Pitman-Yor process weighting structure plate representation.

Geometric process

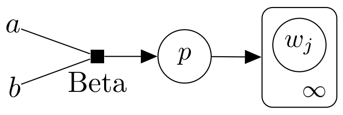

Figure 10: Geometric process weighting structure plate representation.

Beta in Dirichlet process

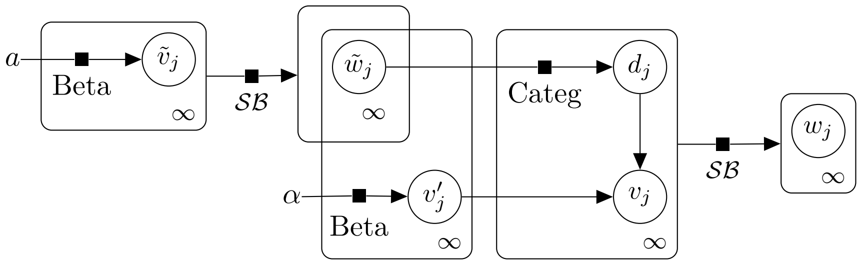

Figure 11: Beta In Dirichlet process weighting structure plate representation.

Beta in beta process

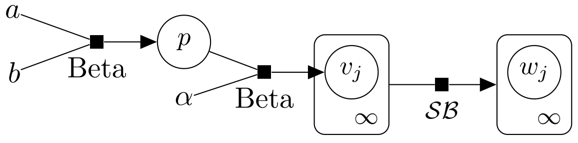

Figure 12: Beta In beta process weighting structure plate representation.

Beta binomial process

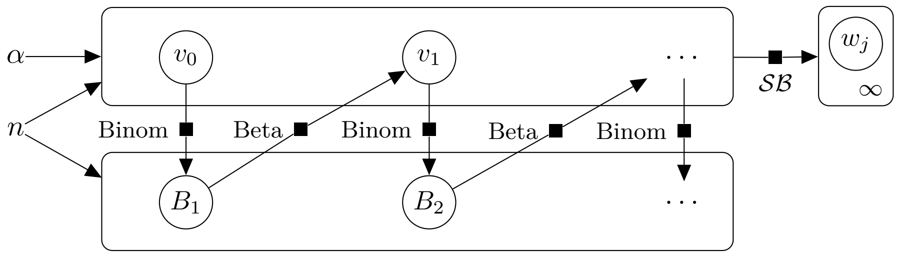

Figure 13: Beta binomial process weighting structure plate representation.

D. Package comparison code

Here we present the R code used to fit the `palmerspenguins` dataset using each of the discussed packages followed by the Python code needed to fit the corresponding packages and the parsing and plotting of results.

R code

```
R> install.packages(c("palmerpenguins", "BNPmix", "mclust",
+    "dirichletprocess", "bayesm", "PReMiuM"),
+    repos = "https://cran.r-project.org")
R> setwd("temp")
R> set.seed(1)
R> library(palmerpenguins)
R> df <- penguins[complete.cases(penguins[3:6]), 3:6]
R> library(BNPmix)
R> bnpmix_time <- system.time({
+    est_model <- PYdensity(
+      y = as.matrix(sweep(df, 2, colMeans(df))),
+      mcmc = list(niter = 1000, nburn = 100, nupd = 1000))
+    out <- partition(est_model)})
R> bnpmix_clustering <- out$partitions[3,]
R> library(mclust)
R> mclust_time <- system.time(
+    out <- Mclust(df, modelNames="VVV"))
R> mclust_clustering <- out$classification
R> library(dirichletprocess)
R> dirichletprocess_time <- system.time({
+    out <- DirichletProcessMvnormal(scale(df))
+    dpCluster <- Fit(out, 100, progressBar = F)})
R> dirichletprocess_clustering <- dpCluster$clusterLabels
R> library(bayesm)
R> bayesm_time <- system.time({
R> out <- rnmixGibbs(Data = list(y = data.matrix(df)),
+    Prior = list(ncomp = 3, a = c(rep(1, 3))),
+    Mcmc = list(R = 1000, keep = 1))
R> outclusterMix <- clusterMix(out$nmix$zdraw[100:1000,])})
R> bayesm_clustering <- outclusterMix$clustera
R> library(PReMiuM)
R> premium_time <- system.time({
+    runInfoObj <- profRegr(covNames = colnames(df), data = df,
+      excludeY = TRUE, xModel = "Normal", seed = 1)
+    disSimObj <- calcDissimilarityMatrix(runInfoObj)
+    out <- calcOptimalClustering(disSimObj, maxNClusters = 3)})
R> premium_clustering <- out$clustering
R> df_clusters <- data.frame(
+    BNPmix = bnpmix_clustering,
+    mclust = mclust_clustering,
+    dirichletprocess = dirichletprocess_clustering,
+    bayesm = bayesm_clustering,
+    PReMiuM = premium_clustering)
R> df_times <- data.frame(
+    BNPmix = bnpmix_time,
+    mclust = mclust_time,
+    dirichletprocess = dirichletprocess_time,
+    bayesm = bayesm_time,
+    PReMiuM = premium_time)
R> write.csv(df_clusters, "R_clusters.csv")
R> write.csv(df_times, "R_times.csv")
```

Python code

```
>>> import time
>>> def get_time_and_clustering(model):
...     start_time = time.time()
...     model.fit_gibbs(y, init_groups=3)
...     group = model.gibbs_map_cluster(y)
...     end_time = time.time()
...     return end_time - start_time, group
>>> times_and_clusters = {
...     "DDM": get_time_and_clustering(
...       pyrichlet.mixture_models.DirichletDistributionMixture(n=3, rng=rng)),
...     "DPM": get_time_and_clustering(
...       pyrichlet.mixture_models.DirichletProcessMixture(rng=rng)),
...     "PYM": get_time_and_clustering(
...       pyrichlet.mixture_models.PitmanYorMixture(pyd=0.1, rng=rng)),
...     "GPM": get_time_and_clustering(
...       pyrichlet.mixture_models.GeometricProcessMixture(rng=rng)),
...     "BIDM": get_time_and_clustering(
...       pyrichlet.mixture_models.BetaInDirichletMixture(a=0.1, rng=rng)),
...     "BIBM": get_time_and_clustering(
...       pyrichlet.mixture_models.BetaInBetaMixture(rng=rng)),
...     "BBM": get_time_and_clustering(
...       pyrichlet.mixture_models.BetaBinomialMixture(rng=rng)),
...     "EWM": get_time_and_clustering(
...       pyrichlet.mixture_models.EqualWeightedMixture(n=3, rng=rng)),
...     "FWM": get_time_and_clustering(
...       pyrichlet.mixture_models.FrequencyWeightedMixture(n=3, rng=rng))
... }
>>> from sklearn.mixture import BayesianGaussianMixture
>>> start_time = time.time()
>>> bgm = BayesianGaussianMixture(n_components=3, max_iter=1000)
>>> sklearn_clustering = bgm.fit_predict(y)
>>> end_time = time.time()
>>> times_and_clusters["sklearn"] = (end_time - start_time,
...   sklearn_clustering)
>>> from mixes import GMM
>>> start_time = time.time()
>>> gmm = GMM(3, num_iter=1000)
>>> gmm.fit(y)
>>> mixes_clustering = gmm.predict(y)
>>> end_time = time.time()
>>> times_and_clusters["mixes"] = (end_time - start_time, mixes_clustering)
>>> from bayesmixpy import run_mcmc
>>> dp_params = """
... fixed_value {
...    totalmass: 1.0
... }
... """
>>> g0_params = f"""
... fixed_values {{
...    mean {{
...      size: {y.shape[1]}"""
>>> for x in y.mean():
...    g0_params += f"""
...      data: {x}"""
>>> g0_params += """
...    }
...    var_scaling: 0.01
...    deg_free: 5
...    scale {
...      rows: 4
...      cols: 4
...      data: 1.0
...      data: 0.0
...      data: 0.0
...      data: 0.0
...      data: 0.0
...      data: 1.0
...      data: 0.0
...      data: 0.0
...      data: 0.0
...      data: 0.0
...      data: 1.0
...      data: 0.0
...      data: 0.0
...      data: 0.0
...      data: 0.0
...      data: 1.0
...      rowmajor: false
...    }
... }
... """
>>> neal2_algo = """
... algo_id: "Neal2"
... rng_seed: 1
... iterations: 1000
... burnin: 100
... init_num_clusters: 1
... """
>>> start_time = time.time()
>>> out = run_mcmc(
...   "NNW", "DP", (y - y.mean()).to_numpy(), g0_params, dp_params,
...   neal2_algo, [], return_clusters=False, return_num_clusters=False,
...   return_best_clus=True)
>>> end_time = time.time()
>>> bayesmix_clustering = out[3]
>>> times_and_clusters["bayesmix"] = (end_time - start_time,
...   bayesmix_clustering)
>>> import subprocess
>>> import pandas as pd
>>> subprocess.call("./article.R")
>>> r_clusters = pd.read_csv('temp/R_clusters.csv', index_col=0)
>>> r_times = pd.read_csv('temp/R_times.csv', index_col=0).loc['elapsed']
>>> times_and_clusters |= {
...    x: (y, z) for x, y, z in
...    zip(r_times.index, r_times.values, r_clusters.T.values)
... }
>>> from sklearn.metrics import mutual_info_score
>>> info_scores = [
...    mutual_info_score(pyrichlet.utils.load_penguins()[1], x[1]) for x in
...    times_and_clusters.values()]
>>> df_times_scores = pd.DataFrame(
...    {'Mutual Information Score': info_scores,
...     'Running Time': [x[0] for x in times_and_clusters.values()]},
...    index=times_and_clusters.keys())
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax2 = ax.twinx()
>>> df_times_scores['Mutual Information Score'].plot(
...    kind='bar', color='#1f77b4', ax=ax, width=0.3, position=1, legend=True)
>>> df_times_scores['Running Time'].plot(
...    kind='bar', color='#ff7f0e', ax=ax2, width=0.3, position=0, legend=True,
...    logy=True)
>>> ax.set_ylabel('Score')
>>> ax.legend(loc='upper left')
>>> ax2.set_ylabel('Seconds')
>>> ax.set_ylim(0, 1.19)
>>> ax2.set_ylim(0.01, 150)
>>> plt.xlim(-1, 17)
>>> plt.tight_layout()
>>> plt.savefig("all_scores.pdf")
>>> plt.show()
>>> def var_get_time_and_clustering(model):
...     start_time = time.time()
...     model.fit_variational(y, n_groups=3)
...     group = model.var_map_cluster(y)
...     end_time = time.time()
...     return end_time - start_time, group
>>> var_times_and_clusters = {
...     "DDM": var_get_time_and_clustering(
...       pyrichlet.mixture_models.DirichletDistributionMixture(n=3, rng=rng)),
...     "DPM": var_get_time_and_clustering(
...       pyrichlet.mixture_models.DirichletProcessMixture(rng=rng)),
...     "PYM": var_get_time_and_clustering(
...       pyrichlet.mixture_models.PitmanYorMixture(pyd=0.1, rng=rng)),
...     "GPM": var_get_time_and_clustering(
...       pyrichlet.mixture_models.GeometricProcessMixture(rng=rng)),
...     "EWM": var_get_time_and_clustering(
...       pyrichlet.mixture_models.EqualWeightedMixture(n=3, rng=rng)),
...     "FWM": var_get_time_and_clustering(
...       pyrichlet.mixture_models.FrequencyWeightedMixture(n=3, rng=rng))
... }
>>> for x in ['sklearn', 'mixes', 'mclust']:
...     var_times_and_clusters[x] = times_and_clusters[x]
>>> var_info_scores = [
...     mutual_info_score(pyrichlet.utils.load_penguins()[1], x[1]) for x in
...     var_times_and_clusters.values()]
>>> df_var_times_scores = pd.DataFrame(
...     {'Mutual Information Score': var_info_scores,
...      'Running Time': [x[0] for x in var_times_and_clusters.values()]},
...     index=var_times_and_clusters.keys())
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax2 = ax.twinx()
>>> df_var_times_scores['Mutual Information Score'].plot(
...     kind='bar', color='#1f77b4', ax=ax, width=0.3, position=1, legend=True)
>>> df_var_times_scores['Running Time'].plot(
...     kind='bar', color='#ff7f0e', ax=ax2, width=0.3, position=0, legend=True,
...     logy=True)
>>> ax.set_ylabel('Score')
>>> ax.legend(loc='upper left')
>>> ax2.set_ylabel('Seconds')
>>> ax.set_ylim(0, 1.19)
>>> ax2.set_ylim(0.01, 15)
>>> plt.xlim(-1, 9)
>>> plt.tight_layout()
>>> plt.savefig("all_var_scores.pdf")
>>> plt.show()
```

Affiliation:

Fidel Selva, Ruth Fuentes-García
Facultad de Ciencias
Uiversidad Nacional Autónoma de México
Ciudad Universitaria S.N.
04510 Mexico City, Mexico
E-mail: `fidel@ciencias.unam.mx`

María Fernanda Gil-Leyva
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Uiversidad Nacional Autónoma de México
Ciudad Universitaria S.N.
04510 Mexico City, Mexico

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
March 2025, Volume 112, Issue 8 *Submitted:* 2022-12-24
doi:10.18637/jss.v112.i08 *Accepted:* 2024-05-02