Mixture Modeling

Michael Betancourt

October 2024

Table of contents

Sometimes a single data generating process just isn't enough. When observations can arise from multiple data generating processes we need to model each of those data generating processes. If we don't know which of those possible data generating processes is responsible for a particular observation then we need to appeal to *mixture modeling*.

Mixture modeling allows us to, for example, account for undesired contamination, such as an irreducible background that overlaps with a desired signal. By mixing together probabilistic models for the signal and background events we can learn not only the behavior of each but also their relative prevalence in the observed data.

In this chapter we'll review the mathematical foundations and general implementation of mixture models, including the potential for frustrating inferential behaviors. Finally we'll illustrate these ideas with a series of demonstrative analyses.

1 Implementing Mixture Models

From a mathematical perspective mixing multiple data generating processes together is relatively straightforward. We have to do a bit of work, however, to derive an implementation that performs well in practice.

1.1 Categorical Implementations

Consider an observational space Y and a collection of K component probability distributions specified with the probability density functions, $p_k(y)$, each modeling a distinct data generating process. To simplify the initial presentation we will assume that each component probability distribution is fixed so that there no model configuration variables to consider.

The most straightforward way to model an observation that can arise from any of these components is to introduce a categorical variable that labels the responsible data generating process,

$$z \in \{1, \ldots, k, \ldots, K\},$$

and a corresponding probabilistic model

$$p(z = k \mid \lambda_1, \ldots, \lambda_K) = \lambda_k$$

where the λ_k from a simplex,

$$0 \leq \lambda_k \leq 1$$
$$\sum_{k=1}^{K} \lambda_k = 1.$$

Each observation is then modeled with a value $y \in Y$, an assignment z, and the full Bayesian model

$$\begin{aligned} p(y, z \mid \lambda_1, \ldots, \lambda_K) &= p(y \mid z)\, p(z \mid \lambda_1, \ldots, \lambda_K) \\ &= p_z(y)\, \lambda_z. \end{aligned}$$

If we know the responsible data generating process then the component assignment variable z will be observed along with y, but if we do not know z then all we can do is infer it from the observed value of y. Assuming that the component probabilities and probability distributions are known the consistent component assignment are given by an application of Bayes' Theorem,

$$\begin{aligned} p(z \mid \tilde{y}, \lambda_1, \ldots, \lambda_K) &= \frac{p(\tilde{y}, z \mid \lambda_1, \ldots, \lambda_K)}{\sum_{k=1}^{K} p(\tilde{y}, k \mid \lambda_1, \ldots, \lambda_K)} \\ &= \frac{p_z(\tilde{y})\, \lambda_z}{\sum_{k=1}^{K} p_k(\tilde{y})\, \lambda_k}. \end{aligned}$$

When modeling multiple, independent observations from overlapping data generating processes we have to consider the heterogeneity of the categorical variables. In particular if all of the categorical variables are modeled with the same component probabilities then the joint full Bayesian model becomes

$$\begin{aligned} p(y_1, z_1, \ldots, y_N, z_N \mid \lambda_1, \ldots, \lambda_K) &= \prod_{n=1}^{N} p(y_n \mid z_n)\, p(z_n \mid \lambda_1, \ldots, \lambda_K) \\ &= p_{z_n}(y_n)\, \lambda_{z_n}. \end{aligned}$$

On the other hand if the component probabilities can vary across observations then we need a separate simplex for each observation,

$$\lambda_n = (\lambda_{n,1}, \ldots, \lambda_{n,K}),$$

and the joint full Bayesian model becomes

$$\begin{aligned} p(y_1, z_1, \ldots, y_N, z_N \mid \lambda_1, \ldots, \lambda_N) &= \prod_{n=1}^{N} p(y_n \mid z_n)\, p(z_n \mid \lambda_n) \\ &= p_{z_n}(y_n)\, \lambda_{n,z_n}. \end{aligned}$$

In either case inferences of the individual z_n depend on only the observed y_n,

$$p(z_n \mid \tilde{y}_n, \lambda_1, \ldots, \lambda_K) = \frac{p_{z_n}(\tilde{y}_n)\, \lambda_{z_n}}{\sum_{k=1}^{K} p_k(\tilde{y}_n)\, \lambda_k},$$

again conditioned on the component probabilities and component probability distributions.

1.2 Marginal Implementations

In practice unknown categorical variables can quickly become a nuisance. Not only do they introduce a new variable that we have to infer for each observation but also those variables are discrete which limits the available computational tools. Specifically we cannot use gradient-based methods like Hamiltonian Monte Carlo to explore posterior distributions over the component assignments and component probabilities, let alone the configuration of the component probability distributions.

Fortunately marginalizing unknown categorical assignments out of the full Bayesian model is straightforward.

1.2.1 Single Observation

For a single observation marginalizing an unknown component assignment requires summing over the K possible values,

$$\begin{aligned} p(y \mid \lambda_1, \ldots, \lambda_K) &= \sum_{k=1}^{K} p(y, z = k \mid \lambda_1, \ldots, \lambda_K) \\ &= \sum_{k=1}^{K} p(y \mid z = k)\, p(z = k \mid \lambda_1, \ldots, \lambda_K) \\ &= \sum_{k=1}^{K} p_k(y)\, \lambda_k. \end{aligned}$$

In other words all we have to do eliminate an categorical variable is add together all of the the component probability density functions weighted by the component probabilities.

1.2.2 Multiple Homogeneous Observations

Given multiple, independent observations modeled by the same component probabilities we can marginalize each component individually,

$$\begin{aligned} p(y_1, \ldots, y_N \mid \lambda_1, \ldots, \lambda_K) &= \prod_{n=1}^{N} p(y_n \mid \lambda_1, \ldots, \lambda_K) \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} p_k(y_n)\, \lambda_k. \end{aligned}$$

1.2.3 Multiple Heterogeneous Observations

Accounting for heterogeneity in the component probabilities is a bit more cumbersome,

$$\begin{aligned} p(y_1, \ldots, y_N \mid \lambda_1, \ldots, \lambda_N) &= \prod_{n=1}^{N} p(y_n \mid \lambda_n) \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} p_k(y_n)\, \lambda_{n,k}. \end{aligned}$$

Interesting we can sometimes simplify the heterogeneous model even further by marginalizing out not only the individual categorical variables but also the individual simplices! Specifically we need the individual simplices to be independent and identically distributed so that any consistent probabilistic model is of the form

$$p(\lambda_n, \ldots, \lambda_n) = \prod_{n=1}^{N} p(\lambda_n).$$

In this case the expectation value of any component probability is the same for all observations,

$$\begin{aligned} \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N p(\lambda_1, \ldots, \lambda_N)\, \lambda_{n,k} &= \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N \left[\prod_{n=1}^{N} p(\lambda_n) \right] \lambda_{n,k} \\ &= \prod_{n' \neq 1}^{N} \int \mathrm{d}\lambda_{n'} p(\lambda_n) \cdot \int \mathrm{d}\lambda_n\, p(\lambda_n)\, \lambda_{n,k} \\ &= \prod_{n' \neq 1}^{N} 1 \cdot \int \mathrm{d}\lambda_n\, p(\lambda_n)\, \lambda_{n,k} \\ &= \int \mathrm{d}\lambda_n\, p(\lambda_n)\, \lambda_{n,k} \\ &\equiv \omega_k. \end{aligned}$$

Moreover these individual expectation values form their own simplex: the boundedness of each $\lambda_{n,k}$ implies that

$$0 \leq \omega_k \leq 1$$

while

$$\begin{aligned}
\sum_{k=1}^{K} \omega_k &= \sum_{k=1}^{K} \int \mathrm{d}\lambda_n \, p(\lambda_n) \, \lambda_{n,k} \\
&= \int \mathrm{d}\lambda_n \, p(\lambda_n) \, \sum_{k=1}^{K} \lambda_{n,k} \\
&= \int \mathrm{d}\lambda_n \, p(\lambda_n) \, 1 \\
&= 1.
\end{aligned}$$

Under these assumptions the marginal mixture model is given by

$$\begin{aligned}
p(y_1, \ldots, y_N) &= \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N p(y_1, \ldots, y_N, \lambda_1, \ldots, \lambda_N) \\
&= \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N \left[\prod_{n=1}^{N} \sum_{k=1}^{K} p_k(y_n) \, \lambda_{n,k} \right] \prod_{n=1}^{N} p(\lambda_n) \\
&= \prod_{n=1}^{N} \int \mathrm{d}\lambda_n \left[\sum_{k=1}^{K} p_k(y_n) \, \lambda_{n,k} \right] p(\lambda_n) \\
&= \prod_{n=1}^{N} \sum_{k=1}^{K} \left[p_k(y_n) \int \mathrm{d}\lambda_n \, p(\lambda_n) \, \lambda_{n,k} \right] \\
&= \prod_{n=1}^{N} \sum_{k=1}^{K} \left[p_k(y_n) \, \omega_k \right].
\end{aligned}$$

In other words the heterogeneous model reduces to the same form as the homogeneous mixture model only with a new simplex

$$\omega = (\omega_1, \ldots, \omega_K)!$$

The new simplex components ω_k can no longer be interpreted as the probability of any *individual* observation arising from a particular component data generating process. Instead they capture the *proportion* of the ensemble of observations that arises from each component data generating process.

1.3 Numerically Stable Marginal Implementations

Most probabilistic programming languages, for example the Stan modeling language, work not with probability density functions $p(y)$ but rather log probability density functions $\log \circ p(y)$.

Consequently in order to implement a mixture model in these languages we need to evaluate

$$\begin{aligned}
\log \circ p(y_1, \ldots, y_N \mid \lambda_1, \ldots, \lambda_K) &= \log \left(\prod_{n=1}^{N} \sum_{k=1}^{K} \lambda_k p_k(y_n) \right) \\
&= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \lambda_k p_k(y_n) \right) \\
&= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \lambda_k \exp(\log \circ p_k(y_n)) \right) \\
&= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \exp(\log(\lambda_k) + \log \circ p_k(y_n)). \right)
\end{aligned}$$

In other words for each observation we need to exponentiate a term for each component, evaluate the sum of the exponentials, and then apply the natural logarithm function again.

This composite operation is often referred to as the "log sum exp" function,

$$\text{log-sum-exp}(x_1, \ldots, x_K) = \log \left(\sum_{k=1}^{K} \exp(x_k) \right).$$

Implementing this operation on computers is complicated by the limitations of floating point arithmetic. In particular the intermediate terms $\exp(x_k)$ are prone to overflowing to floating point infinity and corrupting the calculation before the final logarithm can calm things down.

That said the properties of logarithms and exponentials allow us to implement the log sum exp operation in a variety of ways. Specifically we can always factor out any one component without affecting the final value,

$$\begin{aligned}
\text{log-sum-exp}(x_1, \ldots, x_K) &= \log \left(\sum_{k=1}^{K} \exp(x_k) \right) \\
&= \log \left(\exp(x_{k'}) + \sum_{k \neq k'} \exp(x_k) \right) \\
&= \log \left(\exp(x_{k'}) \right) \left(1 + \sum_{k \neq k'} \frac{\exp(x_k)}{\exp(x_{k'})} \right) \\
&= \log \left(\exp(x_{k'}) \right) \left(1 + \sum_{k \neq k'} \exp(x_k - x_{k'}) \right) \\
&= \log \exp(x_{k'}) + \log \left(1 + \sum_{k \neq k'} \exp(x_k - x_{k'}) \right) \\
&= x_{k'} + \log \left(1 + \sum_{k \neq k'} \exp(x_k - x_{k'}) \right).
\end{aligned}$$

If we factor out the largest component value then

$$x_{k'} \geq x_{k \neq k'}$$

and the input to each exponential will always be less than or equal to one and the calculation will never encounter floating point overflow. Conveniently most computational libraries provide log sum exp function implementations that automatically factor out the largest value and ensure stable numerical computation. Consequently we need to worry about numerical stability only when implementing operations like these ourselves.

For example in Stan we can avoid any issues by using the built-in `log_sum_exp` function.

```
// Loop over observations
for (n in 1:N) {
  target += log_sum_exp(log(lambda[1]) + foo1_lpdf(y[n]),
                        log(lambda[2]) + foo2_lpdf(y[n]),
                        log(lambda[3]) + foo3_lpdf(y[n]),
                        log(lambda[4]) + foo4_lpdf(y[n]));
}
```

Conveniently this function is also vectorized so we can call it with a single vector argument.

```
// Loop over observations
for (n in 1:N) {
  vector[4] lpds = [ foo1_lpdf(y[n]), foo2_lpdf(y[n]),
                     foo3_lpdf(y[n]), foo4_lpdf(y[n]) ]';
  target += log_sum_exp(log(lambda) + lpds);
}
```

When working with only two components there is also a `log_mix` function that incorporates the mixture probabilities directly.

```
log_mix(theta, foo1_lpdf(y[n]), foo2_lpdf(y[n]))
=
log_sum_exp(log(theta)     + foo1_lpdf(y[n]),
            log(1 - theta) + foo2_lpdf(y[n]))
```

All of this said there is one additional numerical concern relevant to the implementation of mixture models. If any categorical probability vanishes,

$$\lambda_k = 0,$$

then the input to the log sum exp function will overflow to negative infinity,

$$\begin{aligned} x_{k'} &= \log(\lambda_{k'}) + \log \circ p_{k'}(y) \\ &= -\infty + \log \circ p_{k'}(y) \\ &= -\infty. \end{aligned}$$

At the same time negative infinities also arise if any of the component probability density functions vanish,

$$\begin{aligned} x_{k'} &= \log(\lambda_{k'}) + \log \circ p_{k'}(y) \\ &= \log(\lambda_{k'}) + \log(0) \\ &= \log(\lambda_{k'}) - \infty \\ &= -\infty. \end{aligned}$$

Formally any negative infinite input results in a vanishing exponential output

$$exp(x_k) = exp(-\infty) = 0$$

and no contribution to the intermediate sum.

$$\sum_{k=1}^{K} \exp(x_k) = \sum_{k \neq k'} \exp(x_k),$$

Consequently we have, at least in theory, the identity

$$\begin{aligned} &\text{log-sum-exp}(x_1, \ldots, x_{k-1}, x_k = -\infty, x_{k+1}, \ldots, x_K) \\ &\qquad = \text{log-sum-exp}(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_K). \end{aligned}$$

In practice, however, the negative infinity can wreak havoc on the floating point calculations.

To avoid potential numerical complications entirely we need to drop the offending component *before* trying to evaluate the inputs to the log sum exp function. Specifically we compute

$$x_k = \log(\lambda_k) + p_k(y)$$

for only the components with $\lambda_k > 0$ and $p_k(y) > 0$, and then evaluate the log sum exp function on only these well-behaved inputs.

1.4 Sampling From Mixture Models

Although we've done a good bit of work to remove the component assignments from mixture models they are not without their uses. Indeed they make sampling from a mixture model particularly straightforward.

The unmarginalized full Bayesian model

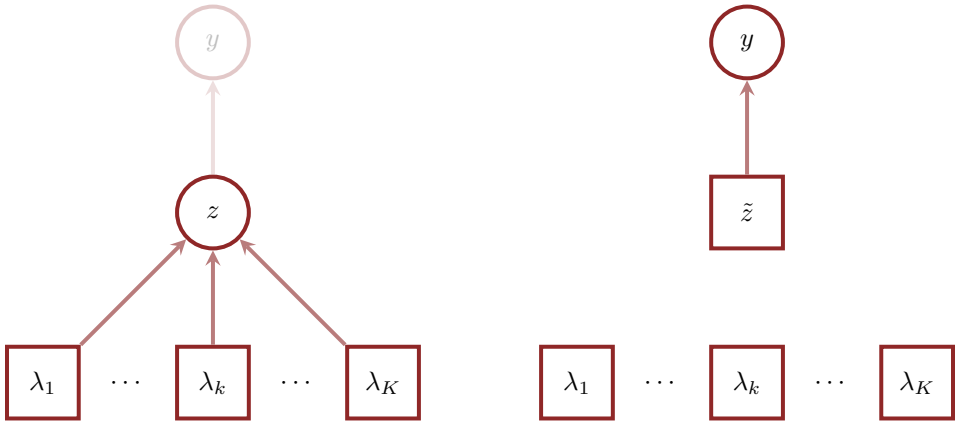
$$p(y, z \mid \lambda_1, \ldots, \lambda_K) = p(y \mid z)\, p(z \mid \lambda_1, \ldots, \lambda_K)$$

immediately motivates an ancestral sampling strategy. We first select a component by sampling a categorical variable (Figure 1a)

$$\tilde{z} \sim p(z \mid \lambda_1, \ldots, \lambda_K)$$

and then sample from the corresponding component probability distribution (Figure 1b)

$$\tilde{y} \sim p(y \mid \tilde{z}) = p_{\tilde{z}}(y).$$



$\tilde{z} \sim p(z \mid \lambda_1, \ldots, \lambda_K)$ $\quad$ $\tilde{y} \sim p(y \mid \tilde{z}) = p_{\tilde{z}}(y)$

(a) $\quad$ (b)

Figure 1: Sampling from a mixture model is a straightforward, two-step process. (a) We first sample a component $\tilde{z}$ given the component probabilities $p(z = k) = \lambda_k$ and then (b) sample an observation $\tilde{y}$ from the corresponding component probability distribution $p_{\tilde{z}}(y)$.

Together the pair $\{\tilde{y}, \tilde{z}\}$ defines a sample from the unmarginalized model,

$$\{\tilde{y}, \tilde{z}\} \sim p(y, z \mid \lambda_1, \ldots, \lambda_K),$$

while the single value $\tilde{y}$ defines a sample from the marginalized model,

$$\tilde{y} \sim p(y \mid \lambda_1, \ldots, \lambda_K).$$

Implementing this two-step sampling procedure in the Stan Modeling Language is straightforward.

```
// Loop over observations
for (n in 1:N) {
  // Sample a component
  int<lower=1, upper=K> z = categorical_rng(lambda);
  // Sample an observation
  if (z == 1) {
    y[n] = foo1_rng();
  } else if (z == 2) {
    y[n] = foo2_rng();
  } else if (z == 3) {
    y[n] = foo3_rng();
  } ...
}
```

Given values for the component probabilities and component probability distributions we can also conditionally sample component assignments for any observation using the posterior distribution that we derived in Section 1.1,

$$\tilde{z} \sim \frac{p_z(y)\,\lambda_z}{\sum_{k=1}^{K} p_k(y)\,\lambda_k}.$$

For example we could sample marginal posterior assignments using the `generated quantities` block in a Stan program.

```
// Loop over observations
for (n in 1:N) {
  vector[4] log_ps =   log(lambda[1]) + foo1_lpdf(y[n])
                     + log(lambda[2]) + foo2_lpdf(y[n])
                     + log(lambda[3]) + foo3_lpdf(y[n])
                     + log(lambda[4]) + foo4_lpdf(y[n]);
  simplex[4] ps = softmax(log_ps);
  z[n] = categorical_rng(log_ps);
}
```

2 Inflation Models

Inflation models are a special case of mixture modeling where one or more of the component probability distributions concentrate on single values in the observation space. These singular probability distributions are defined by the allocations

$$\delta_{y_{\text{inf}}}(\mathsf{y}) = \begin{cases} 1, & y_{\text{inf}} \in \mathsf{y}, \\ 0, & y_{\text{inf}} \notin \mathsf{y} \end{cases}.$$

They are are also known as **Dirac probability distributions**, or simply Dirac distributions for short.

The implementation of inflation models, however, requires some care, especially when the Dirac probability distribution is not compatible with the natural reference measure on a space and we cannot construct well-defined probability density functions.

Note that, while inflation models can in theory contain any number of component probability distributions, in this section I will consider only two, one baseline component and one inflated component, to simplify the presentation.

2.1 Discrete Inflation Models

On discrete spaces Y any Dirac probability distribution can be specified with a probability density function relative to the counting measure, in other words a probability mass function,

$$\delta_{y_{\text{inf}}}(y) = \begin{cases} 1, & y = y_{\text{inf}}, \\ 0, & y \neq y_{\text{inf}} \end{cases}.$$

Consequently we can immediately implement a corresponding inflation model using probability mass functions.

For example we can take a baseline probability distribution specified by the probability mass function $p_{\text{baseline}}(y)$ and inflate the value $y = y_{\text{inf}}$ by mixing it with a Dirac probability mass function,

$$p(y) = \lambda \, p_{\text{baseline}}(y) + (1 - \lambda) \, \delta_{y_{\text{inf}}}(y).$$

That said because the Dirac component allocates zero probability to so many elements we have to take care when evaluating the logarithm of this mixed probability mass function. If $y = y_{\text{inf}}$ then

$$\delta_{\text{inf}}(y) = 1$$

and

$$\log \circ \delta_{\text{inf}}(y) = 0.$$

At this point evaluating the log probability mass function is straightforward,

$$\begin{aligned} \log \circ p(y_{\text{inf}}) &= \text{log-sum-exp}(\log(\lambda) + \log \circ p_{\text{baseline}}(y_{\text{inf}}), \\ &\qquad \log(1 - \lambda) + \log \circ \delta_{\text{inf}}(y_{\text{inf}})) \\ &= \text{log-sum-exp}(\log(\lambda) + \log \circ p_{\text{baseline}}(y_{\text{inf}})), \\ &\qquad \log(1 - \lambda) + \log(1)) \\ &= \text{log-sum-exp}(\log(\lambda) + \log \circ p_{\text{baseline}}(y_{\text{inf}})), \\ &\qquad \log(1 - \lambda)). \end{aligned}$$

On the other hand if $y \neq y_{\text{inf}}$ then

$$\delta_{\text{inf}}(y) = 0$$

and

$$\log \circ \delta_{\text{inf}}(y) = -\infty.$$

To avoid numerical issues arising from the negative infinity we need to follow the procedure introduced in Section 1.3, first accounting for the zero,

$$\begin{aligned} p(y) &= \lambda \, p_{\text{baseline}}(y) + (1 - \lambda) \, \delta_{\text{inf}}(y) \\ &= \lambda \, p_{\text{baseline}}(y) + (1 - \lambda) \, 0 \\ &= \lambda \, p_{\text{baseline}}(y), \end{aligned}$$

and only then taking the natural logarithm,

$$\begin{aligned} \log \circ p(y) &= \log \left(\lambda \, p_{\text{baseline}}(y) \right) \\ &= \log(\lambda) + \log \circ p_{\text{baseline}}(y). \end{aligned}$$

In other words to ensure stable numerical calculations in practice we have to invoke conditional statements when implementing the discrete inflation model. For example in Stan we might write

```
// Loop over observations
for (n in 1:N) {
  // Check equality with inflated value
  if (y[n] == y_inf) {
    target += log_sum_exp(log(lambda) + log_p_baseline(y[n]),
                          log(1 - lambda));
  } else {
    target += log(lambda) + log_p_baseline(y[n]);
  }
}
```

2.2 Continuous Inflation Models

Unfortunately Dirac probability distributions are less well-behaved on continuous spaces. The problem is that Dirac probability distributions are not absolutely continuous with respect to the natural reference measures, such as the Lebesgue measure on a given real space. Consequently we cannot define an appropriate probability density function.

A common heuristic for denoting a continuous inflation model uses the Dirac delta function $\delta(y)$. The Dirac delta function is defined by the integral action

$$\int \mathrm{d}y \, \delta(y) f(y) = f(0)$$

but doesn't actually have a well-defined point-wise output. We can then denote a continuous inflation model with a single inflated value y_{inf} as

$$p(y) = \lambda \, p_{\text{baseline}}(y) + (1 - \lambda) \, \delta(y - y_{\text{inf}})$$

but, because we cannot evaluate $\delta(y - y_{\text{inf}})$, we actually cannot evaluate $p(y)$.

In order to formally implement a continuous inflation model we need to break the observational space Y into two pieces, once containing the inflated value y_{inf} and one containing everything else $Y \setminus y_{\text{inf}}$. We can then treat $\{y_{\text{inf}}\}$ as a discrete space equipped with a counting reference measure and and $Y \setminus y_{\text{inf}}$ as a continuous space equipped with, for example, a Lebesgue reference measure.

If π_{baseline} is absolutely continuous with respect to a Lebesgue reference measure on Y then the probability allocated to the inflated value will be zero

$$\pi_{\text{baseline}}(\{y_{\text{inf}}\}) = 0.$$

Consequently the baseline component will effectively define the same probability distribution on Y and $Y \setminus y_{\text{inf}}$, and we can represent both with the same probability density function. In other words for $y \neq y_{\text{inf}}$ the inflation model can be specified by the probability density function

$$p(y) = \lambda \, p_{\text{baseline}}(y)$$

On the other hand for $y = y_{\text{inf}}$ we have to treat the mixture probability density function as a probability mass function,

$$\begin{aligned} p(y_{\text{inf}}) &= \pi(\{y_{\text{inf}}\}) \\ &= +\lambda \, \pi_{\text{baseline}}(\{y_{\text{inf}}\}) + (1 - \lambda) \, \delta(\{y_{\text{inf}}\}) \\ &= +\lambda \, 0 + (1 - \lambda) \, 1 \\ &= (1 - \lambda). \end{aligned}$$

When working with N independent observations it's convenient to first separate the observations into the N_{inf} observations that exactly equal the inflated value,

$$v_1, \ldots, v_{N_{\text{inf}}},$$

and the remaining $N - N_{\text{inf}}$ observations that don't,

$$w_1, \ldots, w_{N - N_{\text{inf}}}.$$

This allows us to simplify the joint probability density function into the form

$$\begin{aligned}
p(y_1, \ldots, y_N) &= \prod_{n=1}^{N} p(y_n) \\
&= \prod_{n=1}^{N_{\text{inf}}} p(v_n) \prod_{n=1}^{N-N_{\text{inf}}} p(w_n) \\
&= \prod_{n=1}^{N_{\text{inf}}} (1-\lambda) \prod_{n=1}^{N-N_{\text{inf}}} \lambda \, p_{\text{baseline}}(w_n) \\
&= (1-\lambda)^{N_{\text{inf}}} \, \lambda^{N-N_{\text{inf}}} \prod_{n=1}^{N-N_{\text{inf}}} p_{\text{baseline}}(w_n) \\
&\propto \text{Binomial}(N_{\text{inf}} \mid N, 1-\lambda) \prod_{n=1}^{N-N_{\text{inf}}} p_{\text{baseline}}(w_n).
\end{aligned}$$

In other words the continuous inflation model completely decouples into a Binomial model for the number of inflated observations and a baseline model for the non-inflated observations! Moreover because these models are independent of each other they can be implemented together or separately *without impacting any of our inferences*.

The intuition here is that when evaluating the mixture model on the inflated value the contribution of the inflating component is always infinitely larger than the contribution from any other components. If we observe y_{inf} then we know that it *had* to have been generated by the inflating component, and if we observe any other value then we know that it *had* to have been generated by another component. Because of this lack of ambiguity we can always separate the inflated and non-inflated values and model them separately without compromising the consistency of our inferences.

3 Mixture Model Inferences

In practice mixture models are typically used in applications where the neither the component probabilities nor the configuration of the component probability distributions is known precisely. Consequently the main inferential challenge is to infer all of these behaviors from the observed data at the same time.

Theoretically we can infer these behaviors jointly with the unknown component assignments,

$$\begin{aligned} p(\mathbf{z}, \lambda, \theta \mid \tilde{\mathbf{y}}) &\propto p(\tilde{\mathbf{y}}, \mathbf{z}, \lambda, \theta) \\ &\propto \left[p(\tilde{\mathbf{y}} \mid \mathbf{z}, \theta) \, p(\mathbf{z} \mid \lambda) \right] p(\lambda) \, p(\theta) \\ &\propto \prod_{n=1}^{N} \left[p_{z_n}(\tilde{y}_n \mid \theta_{z_n}) \, \lambda_{z_n} \right] p(\lambda) \, p(\theta). \end{aligned}$$

That said in practice it is almost always easier to infer the component probabilities and component model configurations directly from the marginalized model,

$$\begin{aligned} p(\lambda, \theta \mid \tilde{\mathbf{y}}) &\propto p(\tilde{\mathbf{y}}, \lambda, \theta) \\ &\propto \left[p(\tilde{\mathbf{y}} \mid \lambda, \theta) \right] p(\lambda) \, p(\theta) \\ &\propto \prod_{n=1}^{N} \left[\sum_{k=1}^{K} p_k(\tilde{y}_n \mid \theta_k) \, \lambda_k \right] p(\lambda) \, p(\theta). \end{aligned}$$

When needed any component assignment can always be recovered from the corresponding posterior distribution,

$$p(z_n = k \mid \tilde{y}_n, \lambda, \theta) = \frac{p_k(\tilde{y}_n \mid \theta_k) \, \lambda_k}{\sum_{k'=1}^{K} p_{k'}(\tilde{y}_n \mid \theta_k) \, \lambda_k}.$$

From a mathematical perspective the behavior of mixture model inferences is relatively straightforward. For example if the configuration of the component models are fixed then any observations that are more consistent with a particular component model,

$$\frac{\mathrm{d}\pi_{k, \theta_k}}{\mathrm{d}\pi_{k', \theta_{k'}}}(\tilde{y}_n) = \frac{p_k(\tilde{y}_n \mid \theta_k)}{p_{k'}(\tilde{y}_n \mid \theta_{k'})} > 1$$

for all $k' \neq k$, will push the posterior distribution to concentrate on larger values of λ_k and smaller values of the remaining $\lambda_{k'}$.

More generally the posterior distribution will have to account for the interactions between the component probabilities λ and the component model configuration parameters θ. When the component models are not too redundant, with each component model responsible for a mostly unique set of behaviors, then then these interactions will manifest as non-degenerate posterior distribution that are straightforward to quantify with tools like Hamiltonian Monte Carlo.

If the component models exhibit substantial redundancy, however, then the resulting posterior inferences will be much less pleasant. The problem is that the mixture model will be able to contort itself in a variety of different ways to match the observed data. This yields complex,

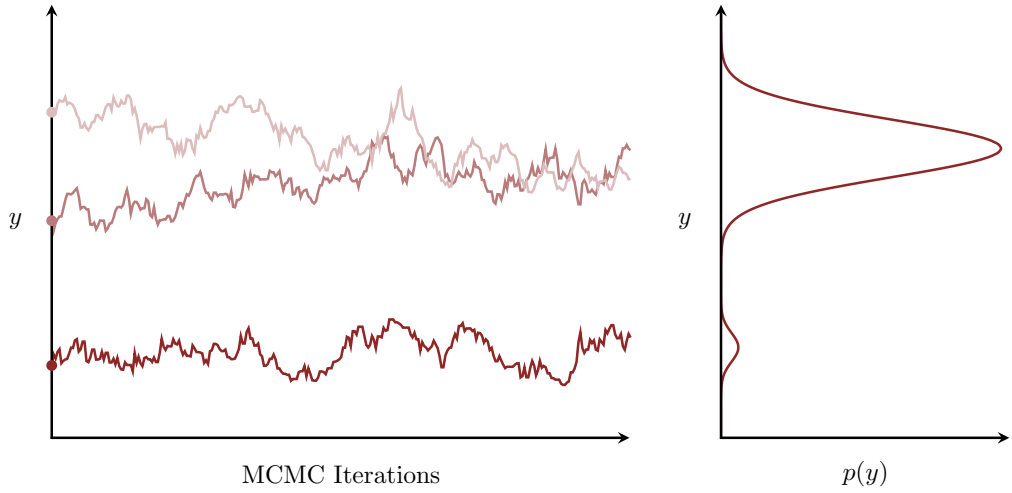


Figure 2: Multi-modal posterior distributions are especially frustrating to Markov chain Monte Carlo. No matter how little probability is allocated to a mode it can still attract Markov chains for arbitrarily long times if they are initialized too close to its basin of attraction.

and often multi-modal, degeneracies that greatly frustrate accurate posterior computation. Recall that multi-modal degeneracies are particularly obstructive as Markov chains with poorly chosen initializations can be trapped by even sub-dominant modes (Figure 2).

Redundancy in a mixture model is at its highest when two or more of the component models are exchangeable, in particular when they span the same family of probability density functions. In this case we can permute the corresponding component indices *without affecting any of the model evaluations*. If K_e component models are exchangeable then the likelihood function will always exhibit $K_e!$ distinct modes, one for each permutation of the redundant indices (Figure 3).

That said non-exchangeable mixture models are not guaranteed to be well-behaved either. Degeneracies can also arise when component models are mathematically distinct but contain similar qualitative behaviors. Consider, for example, a mixture model over positive integers that inflates a baseline Poisson model, $\text{Poisson}(y \mid \lambda)$, with a Dirac probability distribution concentrating at zero, $\delta_0(y)$.

If λ is far from zero then these two components exhibit distinct behaviors, with the former concentrating on values above zero and the latter on values exactly at zero (Figure 4a). When λ is close to zero, however, the two component models will both concentrate at zero (Figure 4b). In this case observations $y = 0$ cannot distinguish between the two components.

In my opinion the most productive approach to mixture modeling is to treat each component

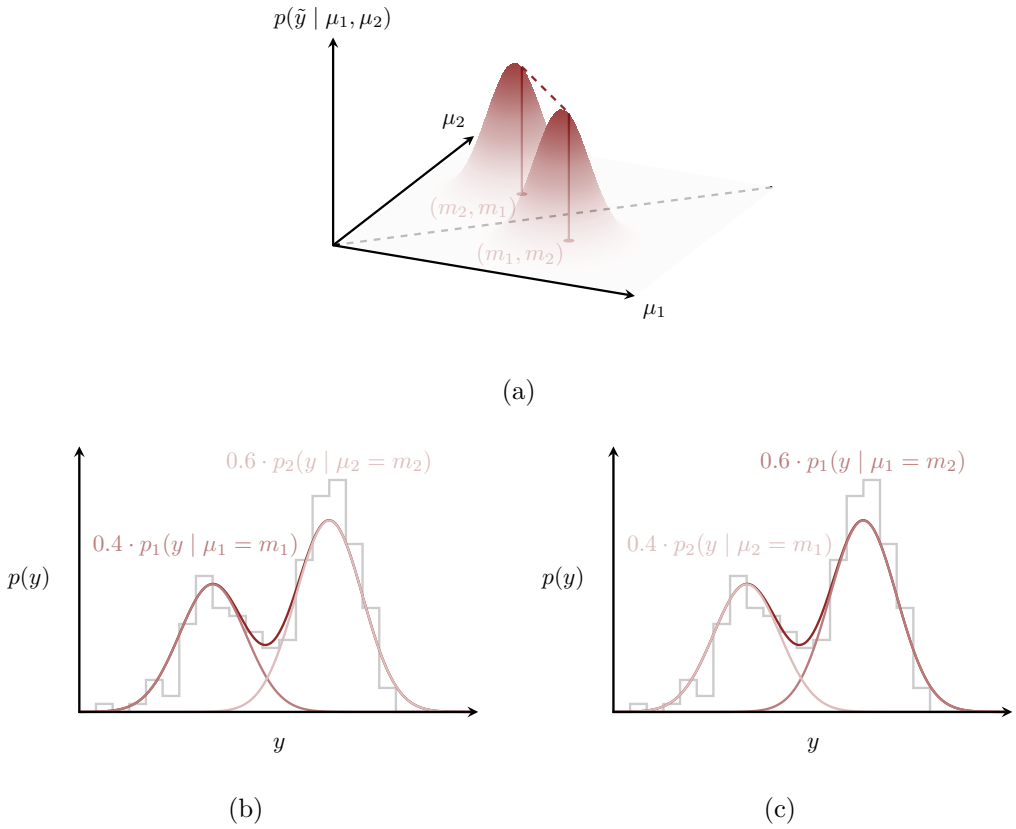
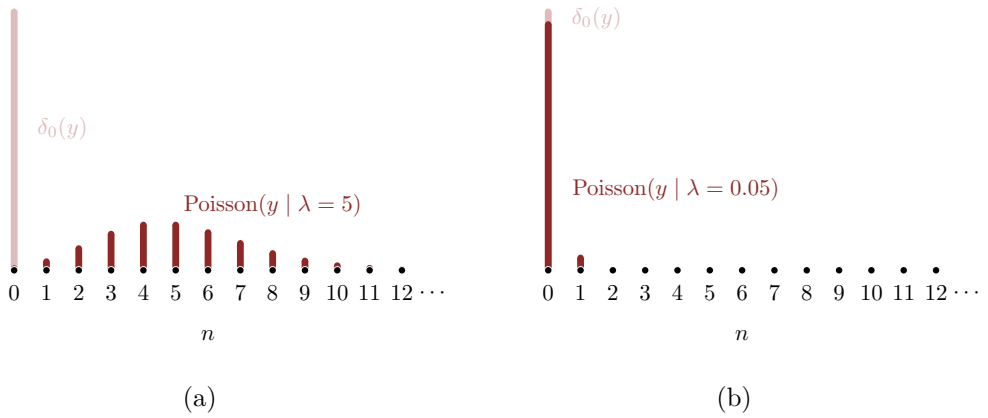


Figure 3: Exchangeable mixture models with K components are formally non-identified – every model configuration is accompanied by $K! - 1$ other model configurations that define exactly the same data generating process, each found in a different mode. For example a mixture model with two identical normal component models always results in (a) a bimodal likelihood function where (b) every model configuration in one mode is paired with (c) a model configuration in another mode where the components are swapped.



(a) (b)

Figure 4: Discrete inflation models are prone to degeneracies when the baseline component can contort itself to mimic the inflated component. (a) When λ is much larger than zero a Poisson model is not easily confused with a Dirac model concentrating at zero. (b) If λ is close to zero, however, then the two component models are nearly indistinguishable.

as a model for a distinct data generating process. Any domain expertise that distinguishes the possible outputs from these data generating processes can then be directly incorporated into either the structure of the component models or the prior model to avoid as much inferential degeneracy as possible.

4 Demonstrations

Enough of the generality. In this section we'll demonstrate the basic mixture modeling techniques presented so in this chapter in specific exercises.

4.1 Setup

Always we start by setting up our local environment.

```
par(family="serif", las=1, bty="l",
    cex.axis=1, cex.lab=1, cex.main=1,
    xaxs="i", yaxs="i", mar = c(5, 5, 3, 1))

library(rstan)
rstan_options(auto_write = TRUE)            # Cache compiled Stan programs
```

```
options(mc.cores = parallel::detectCores()) # Parallelize chains
parallel:::setDefaultClusterOptions(setup_strategy = "sequential")
```

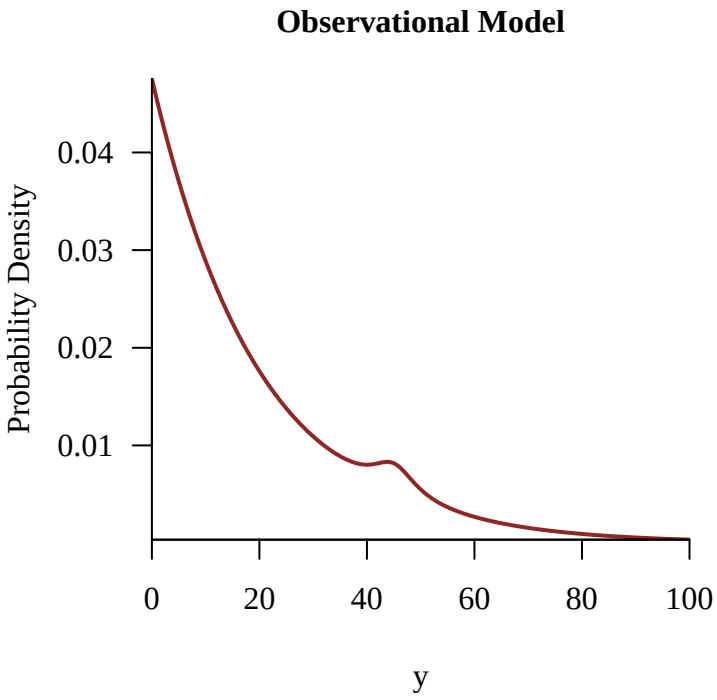
```
util <- new.env()
source('mcmc_analysis_tools_rstan.R', local=util)
source('mcmc_visualization_tools.R', local=util)
```

4.2 Separating Signal and Background

For our first exercise let's consider a simplified version of a particle physics experiment where we observe individual energy depositions in a detector. These source of these depositions can be either an irreducible background or a signal of interest. We'll model the background with a steeply falling exponential probability density function and the signal with a Cauchy probability density function that can be derived from certain physical processes.

Unsurprising given the context of this chapter we can model the overlap of signal and background with a mixture model. The precise configuration of the signal and background models here is arbitrary, especially without units. That said the high background probability is emulative of actual particle physics experiments.

```
mu_signal <- 45
sigma_signal <- 5
beta_back <- 20
lambda <- 0.95

xs <- seq(0, 100, 1)
ys <- lambda * dexp(xs, 1 / beta_back) +
      (1 - lambda) * dcauchy(xs, mu_signal, sigma_signal)

par(mfrow=c(1, 1), mar=c(5, 5, 3, 1))

plot(xs, ys, lwd=2, type="l", col=util$c_dark,
     main="Observational Model",
     xlab="y", ylab="Probability Density")
```



Simulating data from this model is straightforward using the techniques introduced in Section 1.4.

```
N <- 500

simu <- stan(file="stan_programs/simu_signal_background.stan",
             algorithm="Fixed_param",
             data=list("N" = N), seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

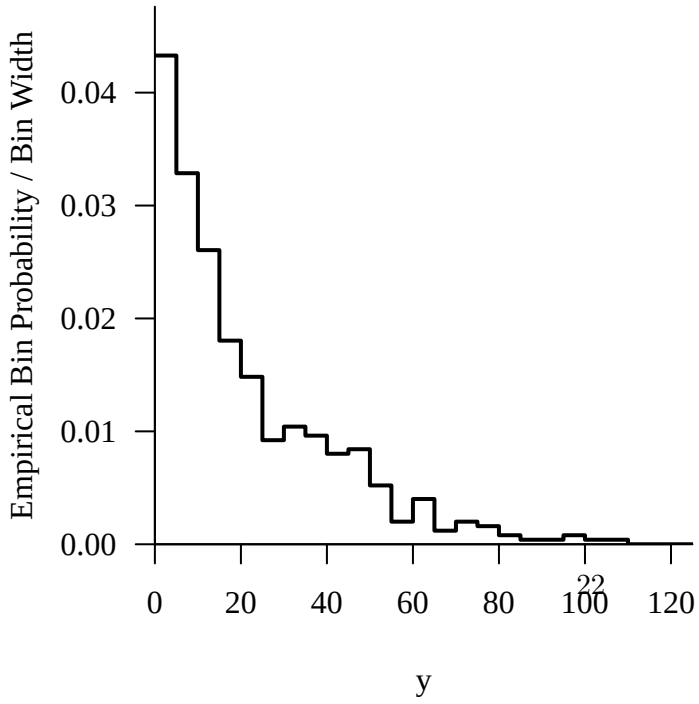
Because of the overwhelming background contribution it's hard to make out the meager signal.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y, 0, 125, 5, xlab="y", prob=TRUE)
```

```
Warning in check_bin_containment(bin_min, bin_max, values): 1 value (0.2%) fell
above the binning.
```

Stan
Program 1 `simu\_signal\_background.stan`

```
data {
  int<lower=1> N;      // Number of observations
}

transformed data {
  real mu_signal =  45;                 // Signal location
  real<lower=0> sigma_signal = 5;       // Signal scale
  real beta_back = 20;                  // Background rate
  real<lower=0, upper=1> lambda = 0.95; // Background probability
}

generated quantities {
  array[N] real<lower=0> y = rep_array(-1, N);

  for (n in 1:N) {
    if (bernoulli_rng(lambda)) {
      y[n] = exponential_rng(1 / beta_back);
    } else {
      // Truncate signal to positive values
      while (y[n] < 0) {
        y[n] = cauchy_rng(mu_signal, sigma_signal);
      }
    }
  }
}
```



Perhaps we can use Bayesian inference to tease out the signal?

Note that the prior model here is somewhat arbitrary due to the nature of the exercise. In a practical analysis we would want to take care to elicit at least some basic domain expertise about the behaviors captured by each parameter. This is especially true if the component models have any potential for redundant behaviors.

```
fit <- stan(file="stan_programs/signal_background1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The diagnostics exhibit some mild $\hat{\xi}$ warnings, but nothing too concerning so long as we're not interested in the expectation value of `mu_signal`.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu_signal', 'sigma_signal',
                                         'beta_back', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

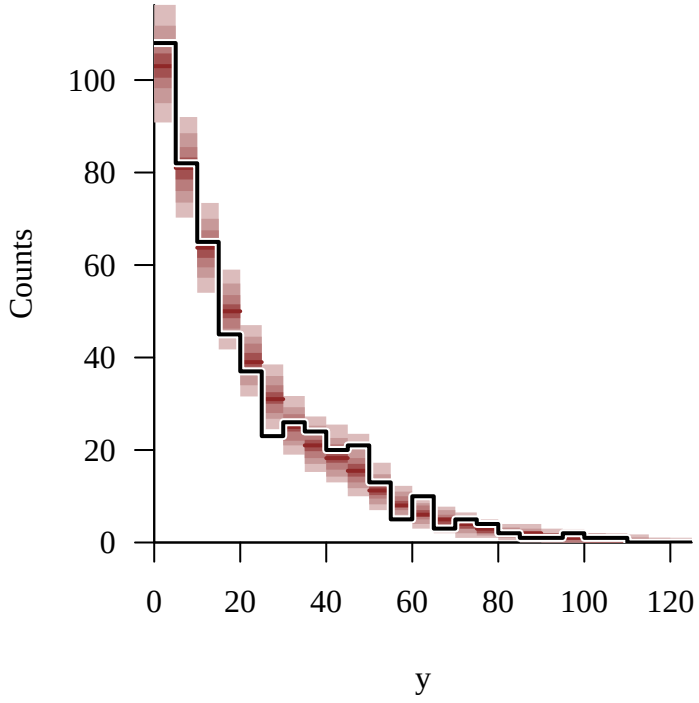
```
mu_signal:
  Chain 1: Right tail hat{xi} (0.253) exceeds 0.25.
  Chain 3: Right tail hat{xi} (0.423) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.378) exceeds 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

Even better there are no indications of retrodictive tension that would suggest inadequacies in our model.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', 0, 125, 5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 6981 predictive values (0.3%) fell above the binning.
```

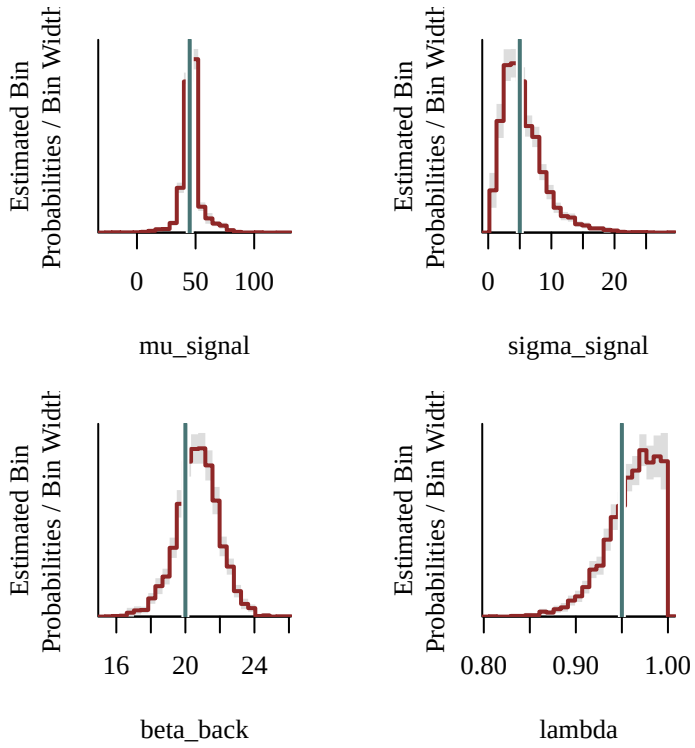
```
Warning in check_bin_containment(bin_min, bin_max, baseline_values, "observed
value"): 1 observed value (0.2%) fell above the binning.
```



Our posterior inferences are not too bad, especially considering the relatively small number of observations and overwhelming background contribution. Note also the heavy tails of marginal posterior distribution for `mu_signal`, consistent with the $\hat{\xi}$ warnings.

```
par(mfrow=c(2, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples[['mu_signal']], 25,
                                display_name="mu_signal",
                                baseline=mu_signal,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['sigma_signal']], 25,
                                display_name="sigma_signal",
                                baseline=sigma_signal,
                                baseline_col=util$c_mid_teal)
```

```
util$plot_expectand_pushforward(samples[['beta_back']], 25,
                                display_name="beta_back",
                                baseline=beta_back,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['lambda']], 25,
                                display_name="lambda",
                                baseline=lambda,
                                baseline_col=util$c_mid_teal)
```



We can also directly visualize the inferred behaviors of the signal model, the background model, and their combinations.

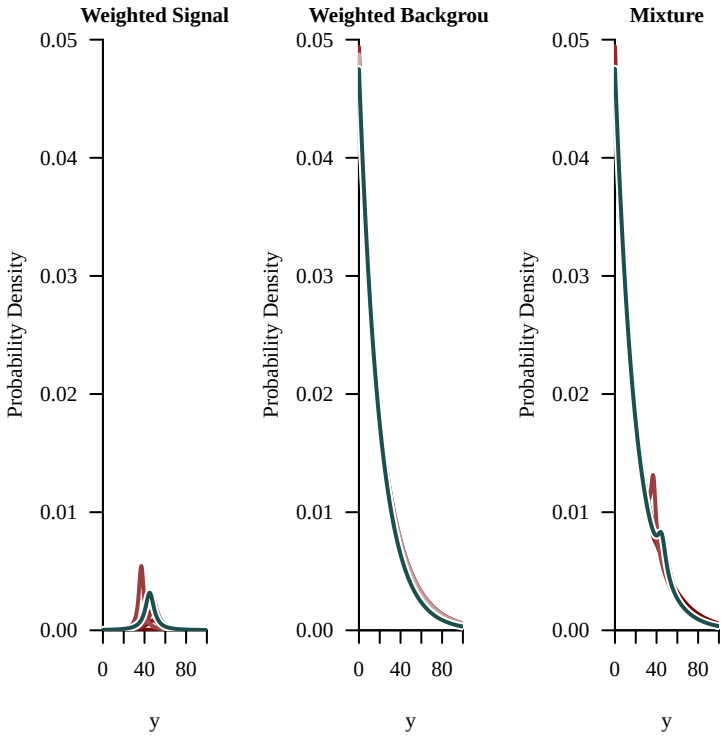
```
library(colormap)
nom_colors <- c("#DCBCBC", "#C79999", "#B97C7C",
                "#A25050", "#8F2727", "#7C0000")
line_colors <- colormap(colormap=nom_colors, nshades=20)

cs <- c(1, 2, 3, 4)
ss <- c(1, 250, 500, 750, 1000)
```

```
plot_signal_realizations <- function() {
  n <- 1
  for (c in cs) {
    for (s in ss) {
      ms <- samples[['mu_signal']][c, s]
      ss <- samples[['sigma_signal']][c, s]
      l <- samples[['lambda']][c, s]

      ys <- (1 - l) * dcauchy(xs, ms, ss)
      lines(xs, ys, lwd=2, col=line_colors[n])
      n <- n + 1
    }
  }
  ys <- (1 - lambda) * dcauchy(xs, mu_signal, sigma_signal)
  lines(xs, ys, lwd=4, col="white")
  lines(xs, ys, lwd=2, col=util$c_dark_teal)
}

plot_back_realizations <- function() {
  n <- 1
  for (c in cs) {
    for (s in ss) {
      bb <- samples[['beta_back']][c, s]
      l <- samples[['lambda']][c, s]

      ys <- l * dexp(xs, 1 / bb)
      lines(xs, ys, lwd=2, col=line_colors[n])
      n <- n + 1
    }
  }
  ys <- lambda * dexp(xs, 1 / beta_back)
  lines(xs, ys, lwd=4, col="white")
  lines(xs, ys, lwd=2, col=util$c_dark_teal)
}

plot_sum_realizations <- function() {
  n <- 1
  for (c in cs) {
    for (s in ss) {
      ms <- samples[['mu_signal']][c, s]
      ss <- samples[['sigma_signal']][c, s]
      bb <- samples[['beta_back']][c, s]
```

```
      l <- samples[['lambda']][c, s]

      ys <- l * dexp(xs, 1 / bb) + (1 - l) * dcauchy(xs, ms, ss)
      lines(xs, ys, lwd=2, col=line_colors[n])
      n <- n + 1
    }
  }
  ys <- (1 - lambda) * dcauchy(xs, mu_signal, sigma_signal) +
        lambda * dexp(xs, 1 / beta_back)
  lines(xs, ys, lwd=4, col="white")
  lines(xs, ys, lwd=2, col=util$c_dark_teal)
}
```

While the posterior uncertainties are large we do seem to be able to resolve the underlying signal through the overwhelming background!

```
par(mfrow=c(1, 3), mar=c(5, 5, 2, 1))

xs <- seq(0, 100, 0.5)

plot(NULL, main="Weighted Signal",
     xlab="y", ylab="Probability Density",
     xlim=range(xs), ylim=c(0, 0.05))
plot_signal_realizations()

plot(NULL, main="Weighted Background",
     xlab="y", ylab="Probability Density",
     xlim=range(xs), ylim=c(0, 0.05))
plot_back_realizations()

plot(NULL, main="Mixture",
     xlab="y", ylab="Probability Density",
     xlim=range(xs), ylim=c(0, 0.05))
plot_sum_realizations()
```



In many scientific applications the relative proportion of signal and background, and hence the component probabilities, are more relevant than the probability that any particular observation arose from either source. That said if the latter is of interest then we computing posterior assignment probabilities and simulating assignments is straightforward.

```
fit <- stan(file="stan_programs/signal_background2.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Because the modified `generated quantities` block consumes pseudo-random numbers differently we need to double check the computational diagnostics. We do see a step adaptation warning which suggests that the adaptation has changed in one of the Markov chains, but on its own this isn't too problematic.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  Chain 1: Averge proxy acceptance statistic (0.676) is
           smaller than 90% of the target (0.801).

  A small average proxy acceptance statistic indicates that the
```

```
adaptation of the numerical integrator step size failed to converge.
This is often due to discontinuous or imprecise gradients.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu_signal', 'sigma_signal',
                                         'beta_back', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

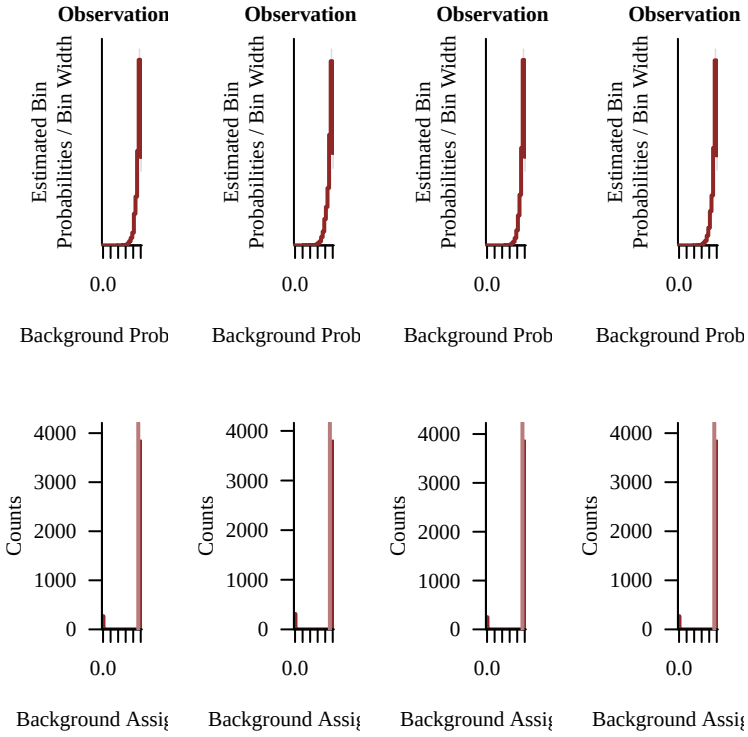
```
mu_signal:
  Chain 1: Right tail hat{xi} (0.516) exceeds 0.25.
  Chain 3: Right tail hat{xi} (0.420) exceeds 0.25.
  Chain 4: Both left and right tail hat{xi}s (0.252, 0.471) exceed 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

Consequently we can move on to analyzing the new behaviors.

```
plot_assignment <- function(idxs) {
  for (idx in idxs) {
    name <- paste0('p[', idx, ']')
    util$plot_expectand_pushforward(samples[[name]], 25,
                                    flim=c(-0.03, 1.03),
                                    display_name='Background Probability',
                                    main=paste("Observation", idx))
  }

  for (idx in idxs) {
    name <- paste0('z_pred[', idx, ']')
    zs <- c(samples[[name]], recursive=TRUE)
    util$plot_line_hist(zs, -0.03, 1.03, 0.02,
                        col=util$c_dark,
                        xlab="Background Assignment")
    abline(v=mean(zs), lwd=2, col=util$c_mid)
  }
}
```

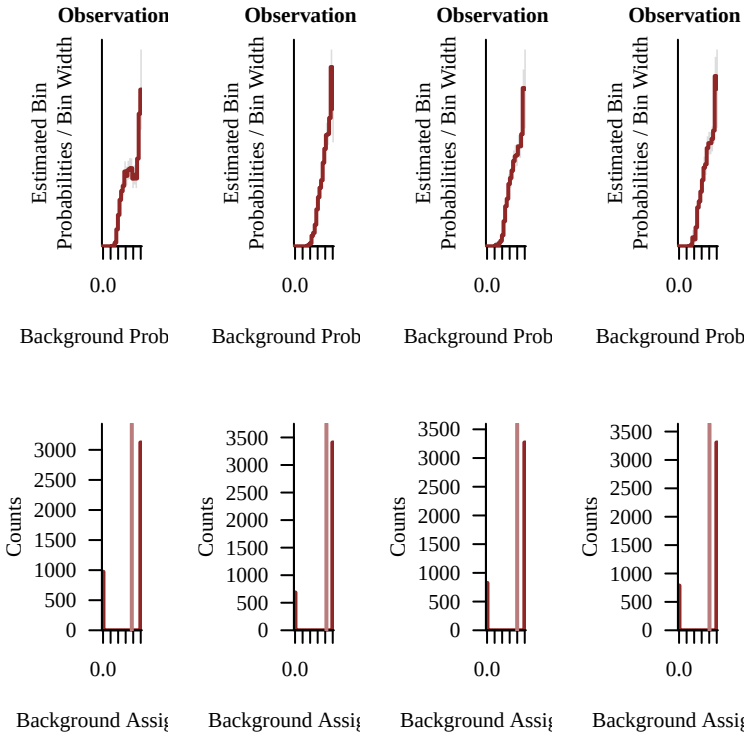
Away from the peak of the signal the observations are strongly associated with the background model, with large individual background probabilities and a predominance of $z = 1$ samples.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

idxs <- which(60 < data$y)
plot_assignment(idxs[1:4])
```



Right at its peak the signal model has more of an influence, although the background model still dominates given its much higher base rate.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

idxs <-  which(45 < data$y & data$y < 55)
plot_assignment(idxs[1:4])
```



In general the component probabilities are always more informative than the sampled assignments. Consequently they are usually the best way to quantify the behavior of individual observations.

4.3 Zero-Inflated Poisson Model

To demonstrate discrete inflation models let's next consider a zero-inflated Poisson model, often affectionately referred to as a "ZIP". Once again we begin by simulating data from a particular configuration of the model.

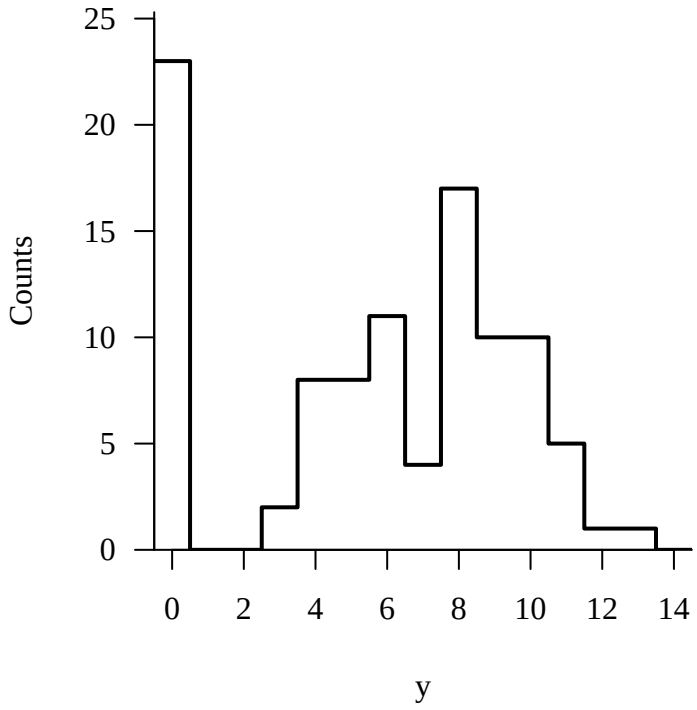
Let's start by simulating data from a configuration where the two component probability distributions are well-separated.

```
N <- 100
mu_true <- 7.5
lambda_true <- 0.8

simu <- stan(file="stan_programs/simu_zip.stan",
             algorithm="Fixed_param",
             data=list("N" = N,
                       "mu" = mu_true,
```

```
                        "lambda" = lambda_true),
             seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

Unsurprisingly we see two clear peaks in the observed data, one concentrating entirely at zero and one scattered across larger values.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y,
                    bin_min=-0.5, bin_max=14.5, bin_delta=1,
                    xlab="y")
```



Because the data so clearly separate into two peaks we might hope that inferences will be straightforward.

```
fit <- stan(file="stan_programs/zip1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Our first good sign is that there are no diagnostics warnings.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

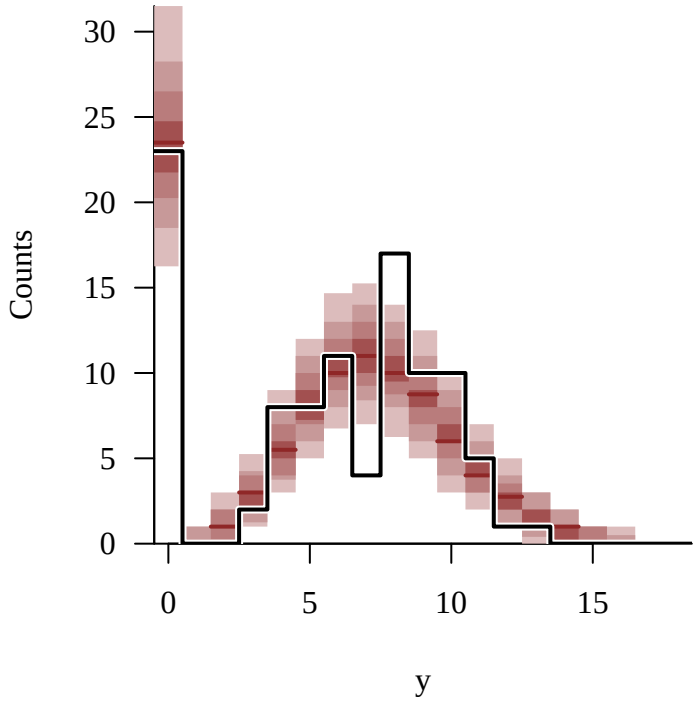
```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples, c('mu', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

A retrodictive check with a histogram summary statistic also looks good. That said retrodictive checks tend to be pretty well-behaved when we're fitting data simulated from the same model!

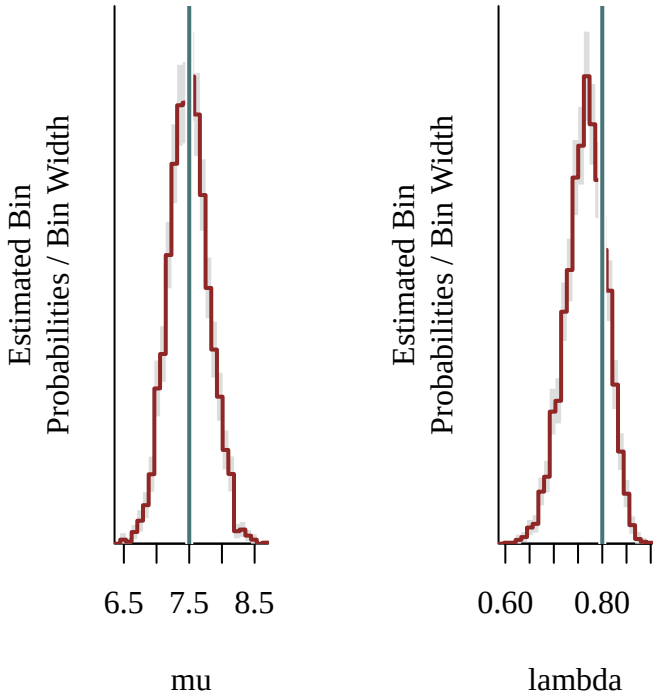
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred',
                         bin_min=-0.5, bin_max=18.5, bin_delta=1,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 99 predictive values (0.0%) fell above the binning.
```



More importantly the posterior inferences are able to identify the true model configuration pretty precisely.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples[['mu']], 25,
                                display_name="mu",
                                baseline=mu_true,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['lambda']], 25,
                                display_name="lambda",
                                baseline=lambda_true,
                                baseline_col=util$c_mid_teal)
```



To make things harder let's try again but with much stronger zero inflation.

```
N <- 100
mu_true <- 7.5
lambda_true <- 0.01

simu <- stan(file="stan_programs/simu_zip.stan",
             algorithm="Fixed_param",
             data=list("N" = N,
                       "mu" = mu_true,
                       "lambda" = lambda_true),
             seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)
```

Indeed the inflation is so strong that the simulated data is comprised entirely of zeros.

```
data <- list("N" = N,
             "y" = extract(simu)$y[1,])

table(data$y)
```

```
  0
100
```

What can we learn about the Poisson component model in this case?

```
fit <- stan(file="stan_programs/zip1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Unfortunately the diagnostics indicate some weak computational problems.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

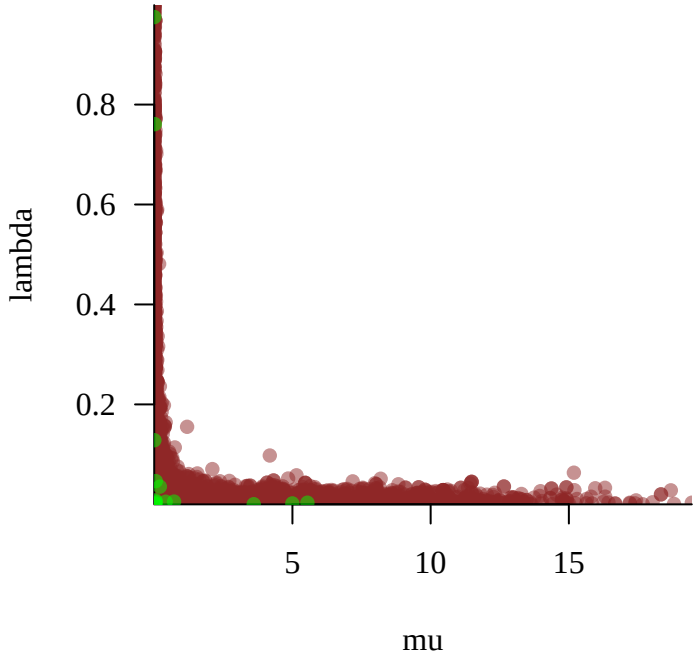
```
  Chain 1: 8 of 1024 transitions (0.8%) diverged.

  Chain 2: 4 of 1024 transitions (0.4%) diverged.

  Chain 4: 2 of 1024 transitions (0.2%) diverged.

  Divergent Hamiltonian transitions result from unstable numerical
trajectories.  These instabilities are often due to degenerate target
geometry, especially "pinches".  If there are only a small number of
divergences then running with adept_delta larger than 0.801 may reduce
the instabilities at the cost of more expensive Hamiltonian
transitions.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples, c('mu', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

```
lambda:
  Chain 1: Right tail hat{xi} (0.899) exceeds 0.25.
  Chain 2: Right tail hat{xi} (0.892) exceeds 0.25.
  Chain 3: Right tail hat{xi} (1.036) exceeds 0.25.
  Chain 4: Right tail hat{xi} (1.222) exceeds 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

Following best practices we'll follow up on the divergences by examining relevant pairs plots. Fortunately here there's only two parameters and hence one possible pairs plot to consider.

```
util$plot_div_pairs(x_names="mu", y_names="lambda",
                    samples, diagnostics)
```



Beyond the scattered divergences an immediate take away from this plot is the extreme posterior uncertainties. The zero-inflated Poisson model can accommodate zeros in two distinct ways. Firstly it can push λ to zero, turning off the baseline Poisson component but also leaving the intensity parameter μ uninformed beyond the prior model. Secondly it can push μ to zero so that the two component models become redundant, in which case λ becomes uninformed beyond the prior model. Ultimately the zero-inflated Poisson model with our initial, diffuse prior model is just too flexible to yield well-behaved inferences.

One way to avoid these strong uncertainties, and the resulting strain on our posterior computation, is to constrain the mixture model with additional domain expertise. For example any information on the strength of the inflation can inform a more concentrated prior model for λ. Similarly any information on the precise value of μ may be able to avoid configurations where the two component models overlap.

Here let's assume that our domain expertise is inconsistent with values of μ below one. The only problem is that we now need a prior model for μ that suppresses values both above 15 and below 1. Multiple families of probability density functions are applicable here, including the log normal, gamma, and inverse gamma families.

All of these families feature slightly different tail behaviors that have different advantages and disadvantages. For this analysis we'll go with the inverse gamma family as it more heavily suppresses the smaller values of μ where we know the component models become redundant.

To inform a particular inverse gamma configuration we can use Stan's algebraic solver to find the configuration matching the desired tail behaviors.

```
stan(file='stan_programs/prior_tune.stan',
     data=list("y_low" = 1, "y_high" = 15),
     iter=1, warmup=0, chains=1,
     seed=4838282, algorithm="Fixed_param")
```

```
alpha = 3.48681
beta = 9.21604

SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 1).
Chain 1: Iteration: 1 / 1 [100%]  (Sampling)
Chain 1:
Chain 1:  Elapsed Time: 0 seconds (Warm-up)
Chain 1:                0 seconds (Sampling)
Chain 1:                0 seconds (Total)
Chain 1:

Inference for Stan model: anon_model.
1 chains, each with iter=1; warmup=0; thin=1;
post-warmup draws per chain=1, total post-warmup draws=1.

      mean se_mean sd 2.5%  25%  50%  75% 97.5% n_eff Rhat
alpha 3.49      NA NA 3.49 3.49 3.49 3.49  3.49     0  NaN
beta  9.22      NA NA 9.22 9.22 9.22 9.22  9.22     0  NaN
lp__  0.00      NA NA 0.00 0.00 0.00 0.00  0.00     0  NaN

Samples were drawn using (diag_e) at Wed Oct  9 23:02:06 2024.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

With a more informative prior model in hand let's try again.

```
fit <- stan(file="stan_programs/zip2.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

There is a lone $\hat{\xi}$ warning but at the same time the divergences are gone.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

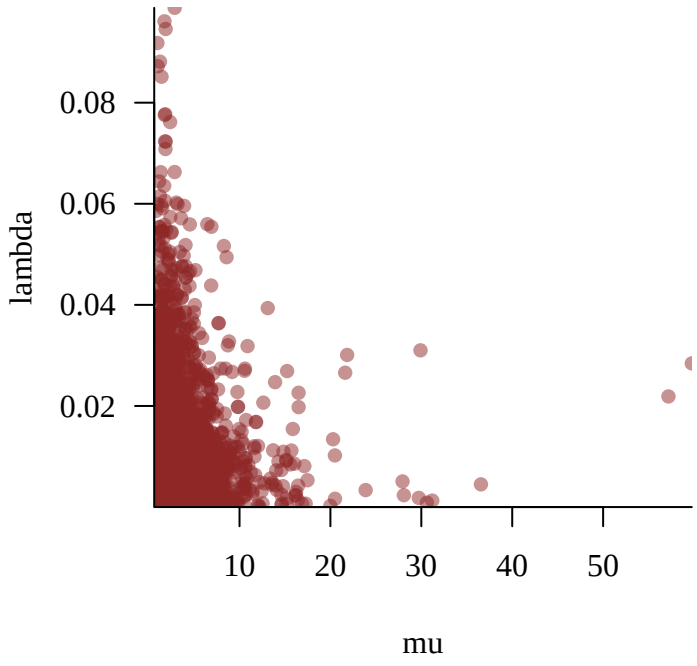
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples, c('mu', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

```
mu:
  Chain 1: Right tail hat{xi} (0.310) exceeds 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

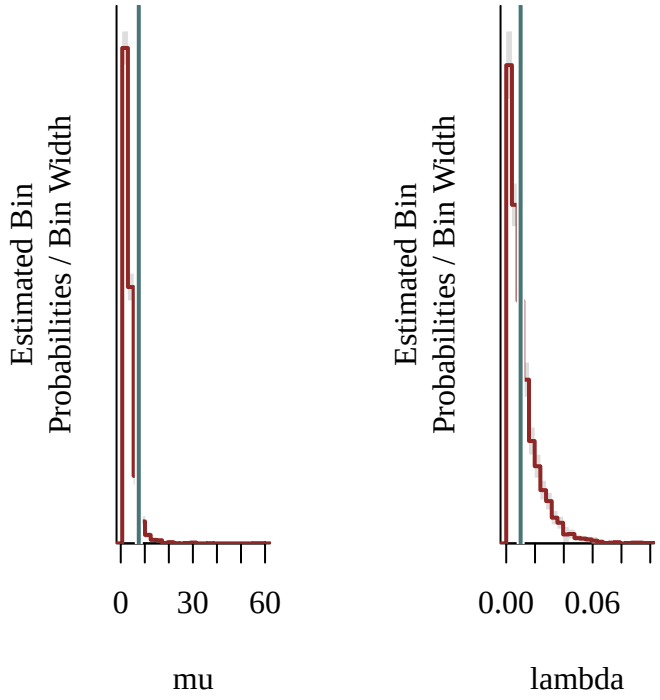
Taking a quick look at the one relevant pairs plot we see that the stronger prior model entirely suppresses the ridge where μ is small and λ is poorly informed.

```
util$plot_div_pairs(x_names="mu", y_names="lambda",
                    samples, diagnostics)
```



That said while the posterior geometry is more well-behaved the inferences still leave much to desired. In particular with this more informative prior model the zero-observations no longer inform μ.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples[['mu']], 25,
                                display_name="mu",
                                baseline=mu_true,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['lambda']], 25,
                                display_name="lambda",
                                baseline=lambda_true,
                                baseline_col=util$c_mid_teal)
```



4.4 Zero/One-Inflated Beta Model

Now let's explore what happens when we try to inflate a continuous baseline model. Continuous inflation models are particularly useful for modeling contamination in a data generating process, such as data-entry errors or data-coding conventions for unexpected or corrupted outcomes. For example data entry software that fills in all entries with zeros before allowing a

user to overwrite that default value will give excess zeros if the user fails to enter all observed values.

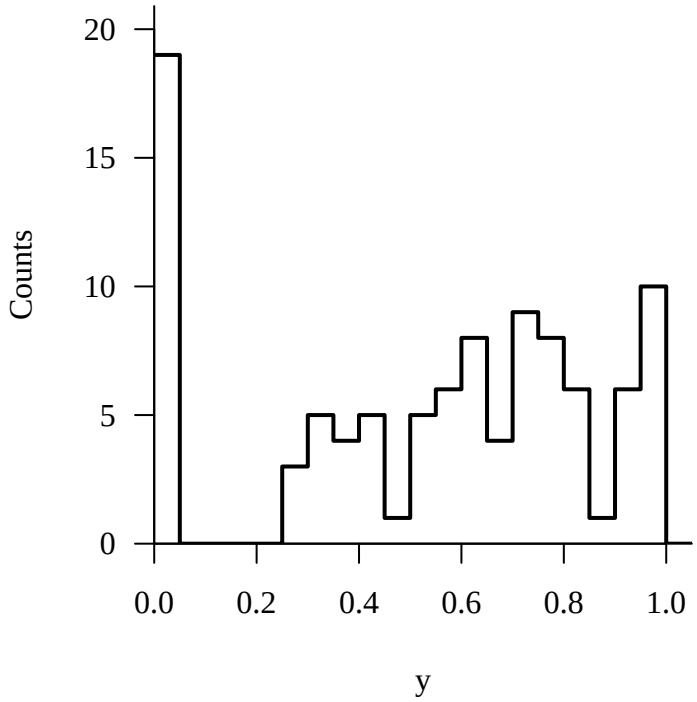
Specifically let's inflate both zero and one values in baseline beta model, giving what is known as a zero/one-inflated beta model. While less conventional than "ZIP" for a zero-inflated Poisson model, the short-hand "ZOIB" for this model is advocated by a small but passionate group.

Simulating data from a continuous inflation model proceeds exactly the same as for a discrete mixture model, and indeed any mixture model.

```
N <- 100

simu <- stan(file="stan_programs/simu_zoib.stan",
             algorithm="Fixed_param",
             data=list("N" = N), seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

Because of the finite binning histogram visualizations of the data can make it difficult to distinguish between inflation *near* zero and one and inflation *exactly at* zero and one.

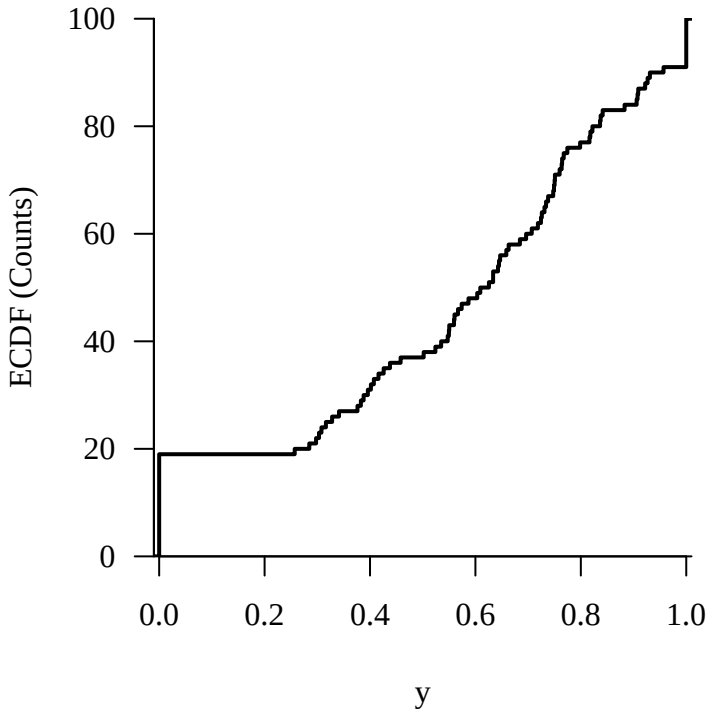
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y,
                    bin_min=0, bin_max=1.05, bin_delta=0.05,
                    xlab="y")
```



Empirical cumulative distribution functions, are better suited to identifying continuous inflation, which manifests as distinct jumps in the cumulative probabilities.

```
plot_ecdf <- function(vals, delta=0.01, xlab="") {
  N <- length(vals)
  ordered_vals <- sort(vals)

  xs <- c(ordered_vals[1] - delta,
          rep(ordered_vals, each=2),
          ordered_vals[N] + delta)

  ecdf_counts <- rep(0:N, each=2)

  plot(xs, ecdf_counts, type="l", lwd="2", col="black",
       xlab=xlab,
       ylim=c(0, N), ylab="ECDF (Counts)")
}
```

```
plot_ecdf(data$y, xlab="y")
```



As discussed in Section 2.2 the implementation of a continuous inflation model requires treating the inflation values as a discrete space separate from the other values. Conveniently there are two ways to handle this.

Firstly we can model the inflated and non-inflated values jointly.

```
fit <- stan(file="stan_programs/zoib1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The diagnostics show no problems.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

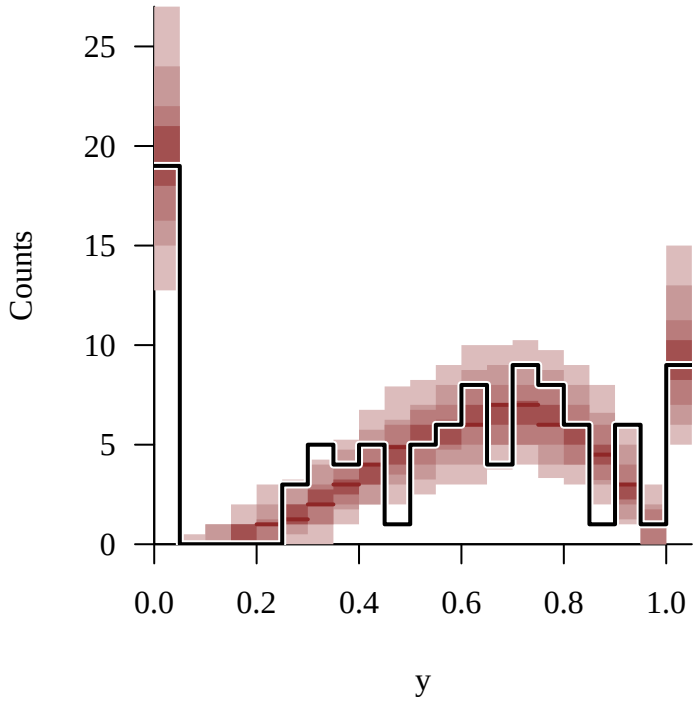
```
samples1 <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples1,
                                       c('alpha', 'beta', 'lambda'),
```

```
                                          check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

Nor does the posterior retrodictive check.

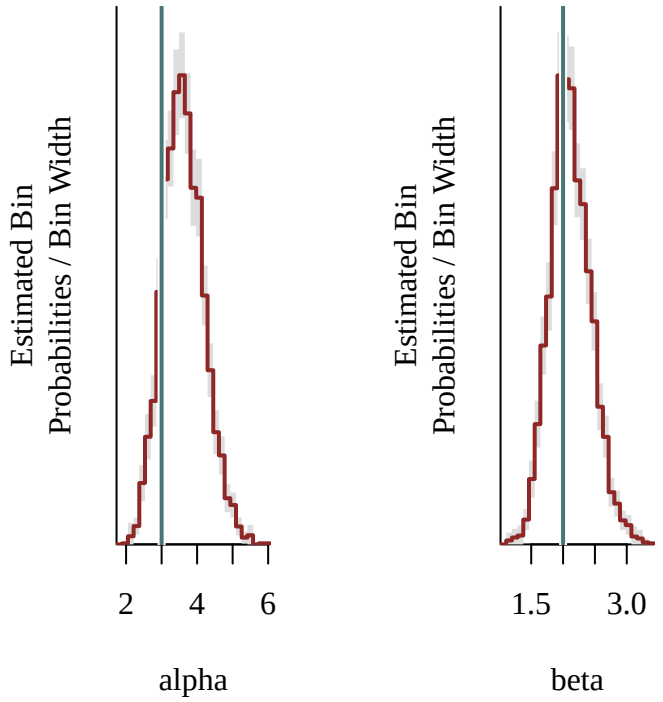
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples1, 'y_pred',
                         bin_min=0, bin_max=1.05, bin_delta=0.05,
                         baseline_values=data$y, xlab="y")
```



Moreover our posterior inferences are both precise and accurate to the simulation data generating process. For instance the inferences are consistent with the true configuration of the baseline beta model used to simulate the data.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples1[['alpha']], 25,
```

```
                                  display_name="alpha",
                                  baseline=3,
                                  baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples1[['beta']], 25,
                                  display_name="beta",
                                  baseline=2,
                                  baseline_col=util$c_mid_teal)
```



Because we have three components we can directly visualize the simplex of component probabilities, instead of for example having to visualize the behavior of each component independently and neglecting the simplex constraints.

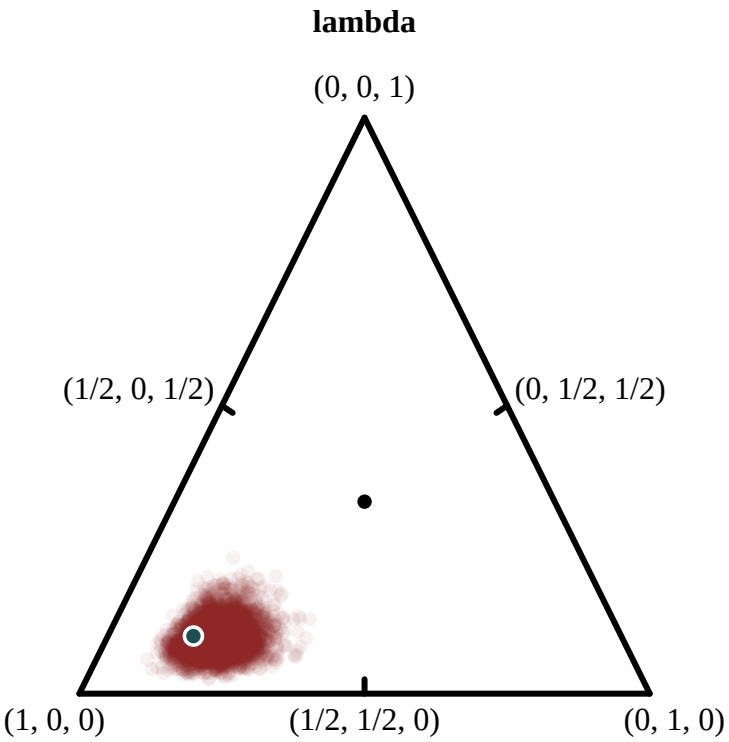
```
to_plot_coordinates <- function(q, C) {
  c(C * (q[2] - q[1]), q[3])
}

plot_simplex_border <- function(label_cex, C, center_label=TRUE) {
  lines( c(-C, 0), c(0, 1), lwd=3)
  lines( c(+C, 0), c(0, 1), lwd=3)
  lines( c(-C, +C), c(0, 0), lwd=3)
```

```
  text_delta <- 0.05
  text( 0, 1 + text_delta, "(0, 0, 1)", cex=label_cex)
  text(-C - text_delta, -text_delta, "(1, 0, 0)", cex=label_cex)
  text(+C + text_delta, -text_delta, "(0, 1, 0)", cex=label_cex)

  tick_delta <- 0.025
  lines( c(0, 0), c(0, tick_delta), lwd=3)
  text(0, 0 - text_delta, "(1/2, 1/2, 0)", cex=label_cex)

  lines( c(+C * 0.5, +C * 0.5 - tick_delta * 0.5 * sqrt(3)),
         c(0.5, 0.5 - tick_delta * 0.5), lwd=3)
  text(C * 0.5 + text_delta * 0.5 * sqrt(3) + 2.5 * text_delta,
       0.5 + text_delta * 0.5, "(0, 1/2, 1/2)", cex=label_cex)

  lines( c(-C * 0.5, -C * 0.5 + tick_delta * 0.5 * sqrt(3)),
         c(0.5, 0.5 - tick_delta * 0.5), lwd=3)
  text(-C * 0.5 - text_delta * 0.5 * sqrt(3) - 2.5 * text_delta,
       0.5 + text_delta * 0.5, "(1/2, 0, 1/2)", cex=label_cex)

  points(0, 1/3, col="white", pch=16, cex=1.5)
  points(0, 1/3, col="black", pch=16, cex=1)
  if (center_label)
    text(0, 1/3 - 1.5 * text_delta, "(1/3, 1/3, 1/3)", cex=label_cex)
}

plot_simplex_samples <- function(q1, q2, q3, label_cex=1,
                                 main="", baseline=NULL) {
  N <- 200
  C <- 1 / sqrt(3)

  plot(NULL, xlab="", ylab="", xaxt="n", yaxt="n", frame.plot=F,
       xlim=c(-(C + 0.2), +(C + 0.2)), ylim=c(-0.1, 1.1))
  plot_simplex_border(label_cex, C, FALSE)
  title(main)

  N <- min(length(q1), length(q2), length(q3))
  for (n in 1:N) {
    xy <- to_plot_coordinates(c(q1[n], q2[n], q3[n]), C)
    points(xy[1], xy[2], col="#8F272710", pch=16, cex=1.0)
  }
  if (!is.null(baseline)) {
    xy <- to_plot_coordinates(baseline, C)
```

```
    points(xy[1], xy[2], col="white", pch=16, cex=1.5)
    points(xy[1], xy[2], col=util$c_dark_teal, pch=16, cex=1)
  }
}
```

This allows us to clearly see the marginal posterior distribution for the component probabilities concentrating around the true value.

```
par(mfrow=c(1, 1), mar=c(0, 0, 2, 0))

plot_simplex_samples(c(samples1[['lambda[1]']], recursive=TRUE),
                     c(samples1[['lambda[2]']], recursive=TRUE),
                     c(samples1[['lambda[3]']], recursive=TRUE),
                     main="lambda", baseline=c(0.75, 0.15, 0.10))
```



While the joint model works well we can also fit the inflated and non-inflated values independently of each other.

```
data$N_zero <- sum(data$y == 0)
data$N_one  <- sum(data$y == 1)
```

```
data$y_else <- data$y[data$y != 0 & data$y != 1]
data$N_else <- length(data$y_else)
```

```
fit <- stan(file="stan_programs/zoib2a.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)

diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples2a <- util$extract_expectand_vals(fit)
util$check_all_expectand_diagnostics(samples2a)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

```
fit <- stan(file="stan_programs/zoib2b.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)

diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

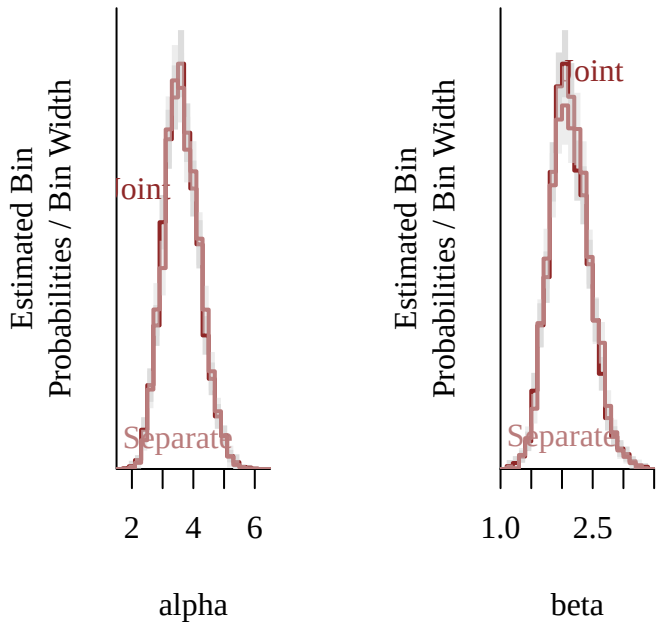
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples2b <- util$extract_expectand_vals(fit)
util$check_all_expectand_diagnostics(samples2b)
```

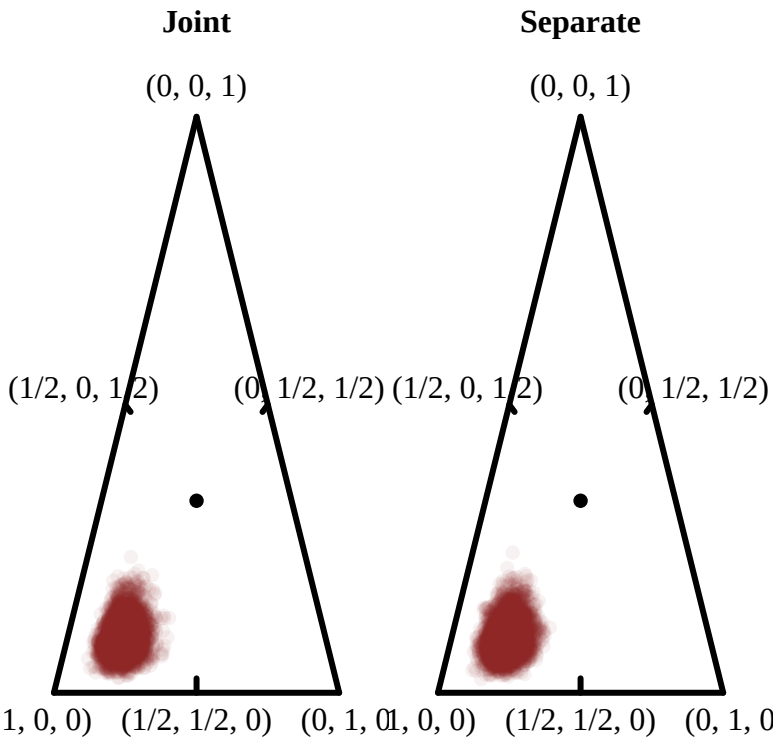
```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

Critically the posterior inferences derived in the two approaches are consistent up to the expected Markov chain Monte Carlo variation.

```
par(mfrow=c(1, 2), mar=c(5, 5, 3, 1))

util$plot_expectand_pushforward(samples1[['alpha']],
                                25, flim=c(1.5, 6.5),
                                display_name="alpha")
text(2.25, 0.5, "Joint", col=util$c_dark)

util$plot_expectand_pushforward(samples2a[['alpha']],
                                25, flim=c(1.5, 6.5),
                                col=util$c_mid,
                                border="#DDDDDD88",
                                add=TRUE)
text(3.5, 0.05, "Separate", col=util$c_mid)

util$plot_expectand_pushforward(samples1[['beta']],
                                25, flim=c(1, 3.5),
                                display_name="beta",)
text(2.5, 1.3, "Joint", col=util$c_dark)

util$plot_expectand_pushforward(samples2a[['beta']],
                                25, flim=c(1, 3.5),
                                col=util$c_mid,
                                border="#DDDDDD88",
                                add=TRUE)
text(2, 0.1, "Separate", col=util$c_mid)
```




```
par(mfrow=c(1, 2), mar=c(0, 0, 2, 0))

plot_simplex_samples(c(samples1[['lambda[1]']], recursive=TRUE),
                     c(samples1[['lambda[2]']], recursive=TRUE),
                     c(samples1[['lambda[3]']], recursive=TRUE),
                     main="Joint")

plot_simplex_samples(c(samples2b[['lambda[1]']], recursive=TRUE),
                     c(samples2b[['lambda[2]']], recursive=TRUE),
                     c(samples2b[['lambda[3]']], recursive=TRUE),
                     main="Separate")
```



If the inflated values are modeling some undesired contamination and we are interested in only the behavior of the continuous baseline model then we can always fit the baseline model to the non-inflated values directly and ignore the inflated values entirely. Similarly if we are interested in only the prevalence of inflated values then we can fit the component probabilities to the counts directly, ignoring the precise value of the continuous observations.

4.5 Exchangeable Mixture Model

To avoid any ambiguity let me make it clear that I do not recommend exchangeable mixture models in applied practice. Without some way of distinguishing the component probability distributions mixture models are inherently prone to degenerate inferences that frustrate meaningful insights.

That said in this section we'll explore an exchangeable mixture of normal component models, if only to see how problematic they can be.

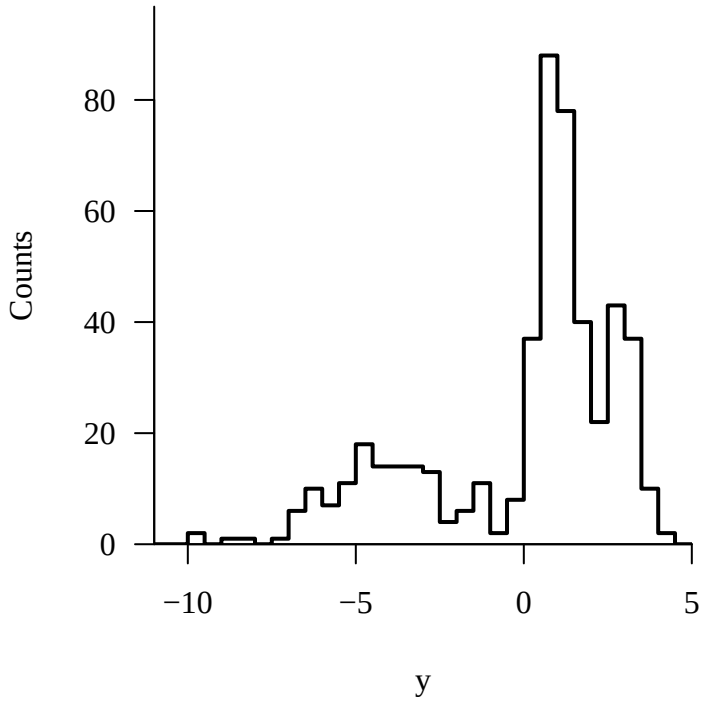
At the very least the simulation of data is straightforward.

```
N <- 500

simu <- stan(file="stan_programs/simu_normal_mix.stan",
             algorithm="Fixed_param",
```

```
              data=list("N" = N), seed=8438338,
              warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
              "y" = extract(simu)$y[1,])
```

A histogram of the data clearly shows at least two peaks, with the possibility of the second peak separating into two peaks of its own.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y, -11, 5, 0.5, xlab="y")
```



4.5.1 Unknown Component Probabilities

Recall that exchangeablity is a property of our knowledge of the system. If we know the true configuration of the component mixture models, so that all we have to infer are the component probabilities, then the mixture model is not actually exchangeable.

```
fit <- stan(file="stan_programs/normal_mix1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

In this case the computation is clean.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

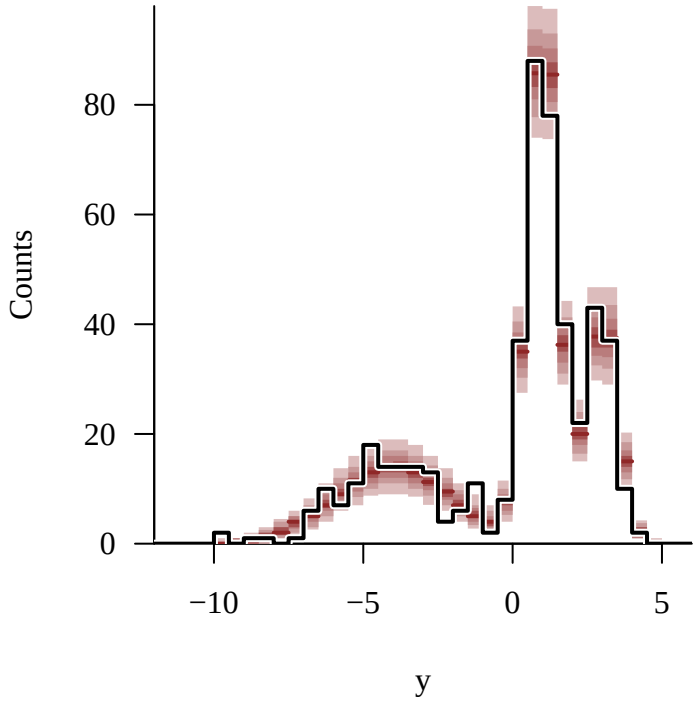
```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

The posterior retrodictive check not only shows no signs of model inadequacy but also shows that our model infers three distinct peaks from the observed data.

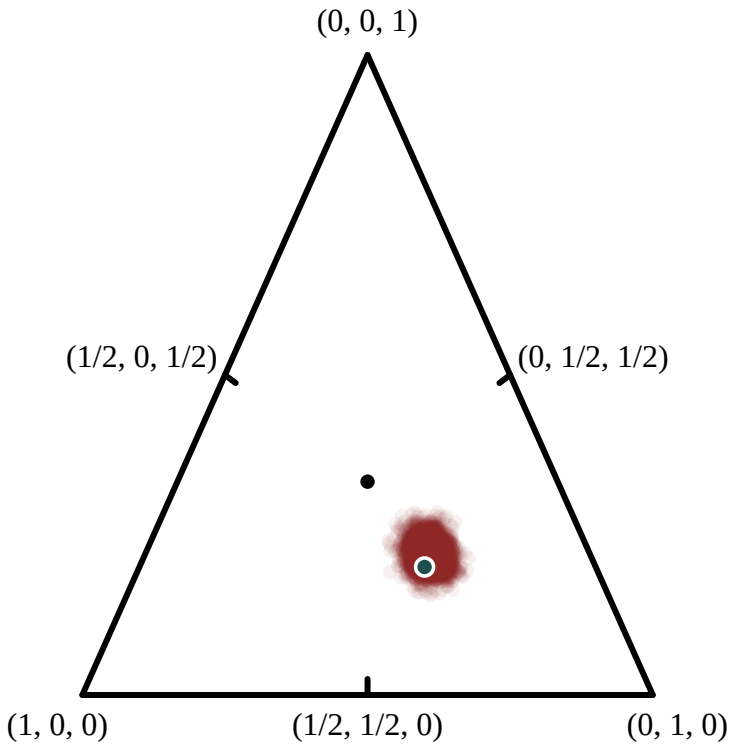
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 12 predictive values (0.0%) fell below the binning.
```



Moreover posterior inferences for the component probabilities are consistent with the true values used to simulate the data.

```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

lambda_true <- c(0.3, 0.5, 0.2)
plot_simplex_samples(c(samples[['lambda[1]']], recursive=TRUE),
                     c(samples[['lambda[2]']], recursive=TRUE),
                     c(samples[['lambda[3]']], recursive=TRUE),
                     baseline=lambda_true)
```



4.5.2 Unknown Component Probabilities and Locations

Let's now complicate the situation by leaving the component scales fixed but trying to infer the component locations. Because the component scales are not all equal to each other the resulting mixture model is not completely exchangeable. That said the entire mixture model is approximately exchangeable, and the latter two components on their own are exactly exchangeable.

```
fit <- stan(file="stan_programs/normal_mix2a.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The preponderance of split $\hat{R}$ warnings hints at a multi-modal posterior distribution.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

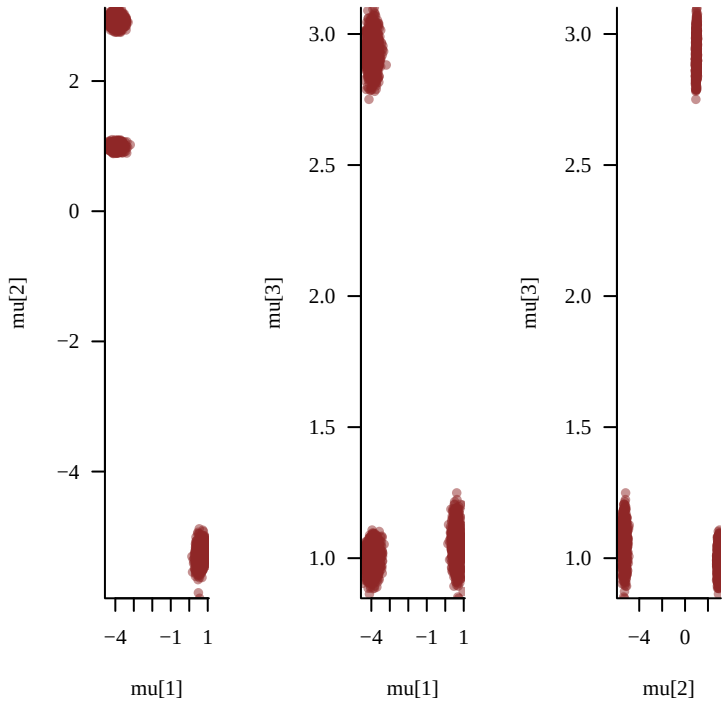
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

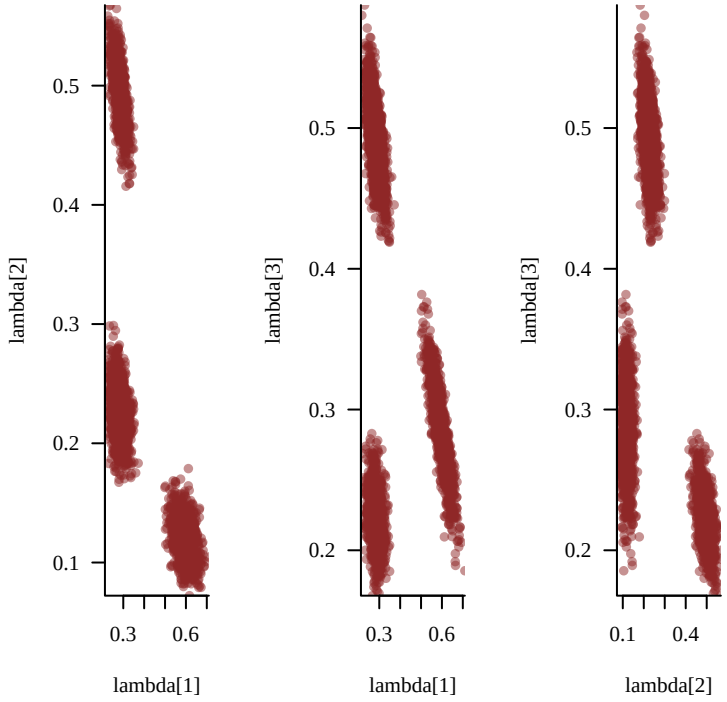
```
mu[1]:
  Split hat{R} (12.002) exceeds 1.1.

mu[2]:
  Split hat{R} (47.876) exceeds 1.1.

mu[3]:
  Split hat{R} (18.290) exceeds 1.1.

lambda[1]:
  Split hat{R} (5.779) exceeds 1.1.

lambda[2]:
  Split hat{R} (7.297) exceeds 1.1.

lambda[3]:
  Split hat{R} (5.198) exceeds 1.1.


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

Indeed the pairs plots show at least three distinct posterior modes, although counting modes based on two-dimensional projections is not always straightforward. Moreover there could easily be more modes that our Markov chains missed.

```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```

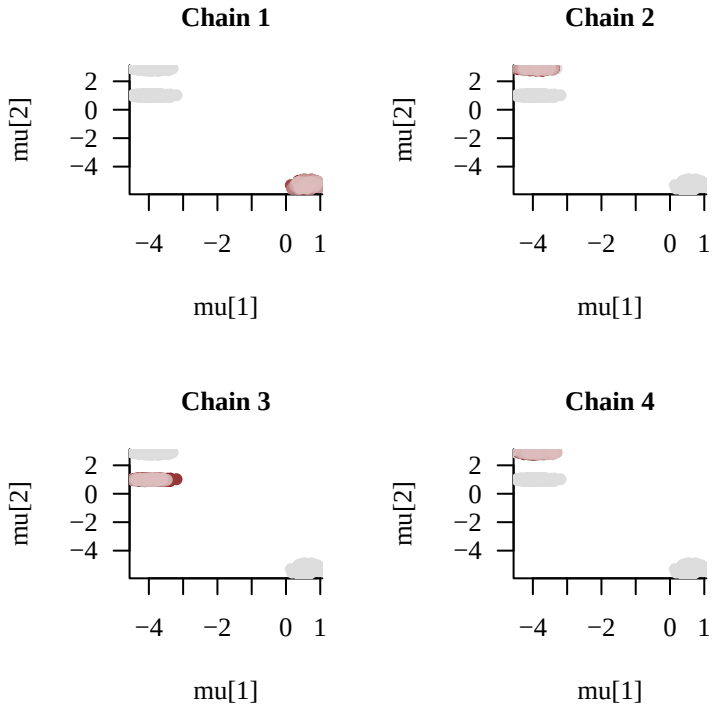



```
names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



Unsurprisingly the individual Markov chains are each confined to a single mode. Because the first, second, and third Markov chains appear to have fallen into distinct modes we'll focus on those going forwards.

```
util$plot_pairs_by_chain(samples[['mu[1]']], 'mu[1]',
                         samples[['mu[2]']], 'mu[2]')
```



The propensity for individual modes to capture entire Markov chains is both why we cannot accurately estimate the relative important of each mode and why running many independent Markov chains is the most practical way to diagnose multi-modality.

To understand the behavior within each mode let's explore the inferred component behaviors within individual Markov chains, and hence within individual modes.

```
plot_component_realizations <- function(k, c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_name <- paste0('mu[', k, ']')
    mu <- samples[[mu_name]][c, s]

    sigma <- c(2, 0.5, 0.5)[k]

    lambda_name <- paste0('lambda[', k, ']')
```

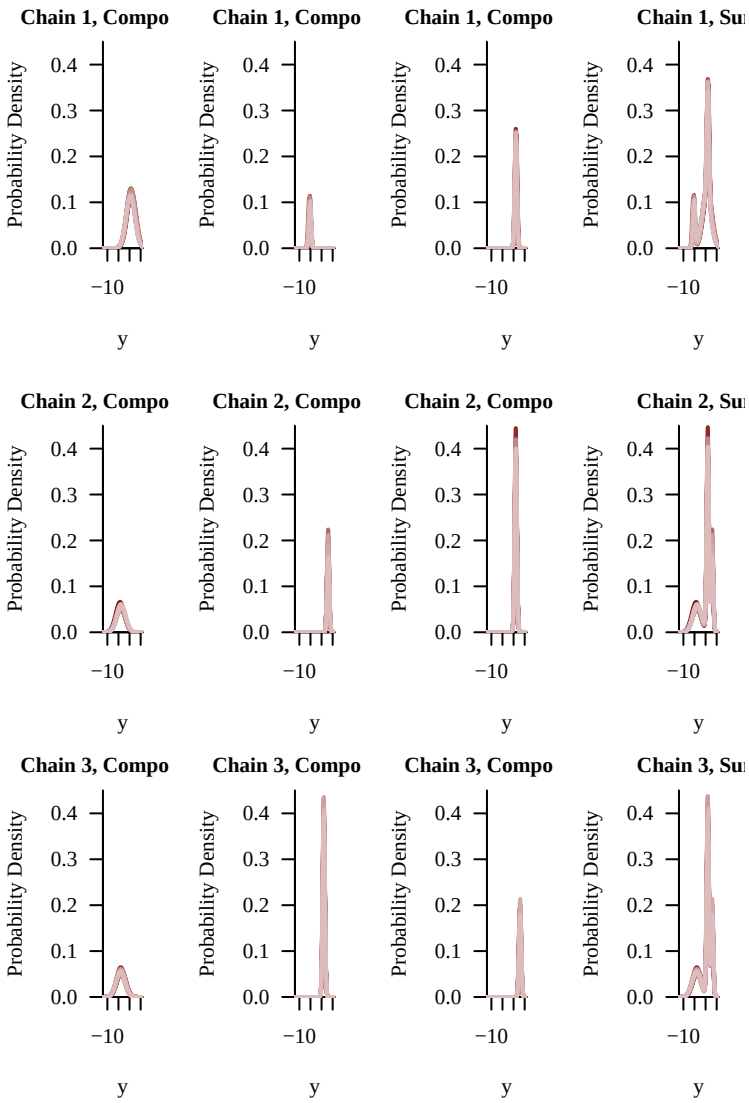
```
    lambda <- samples[[lambda_name]][c, s]

    ys <- lambda * dnorm(xs, mu, sigma)
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
}

plot_sum_realizations <- function(c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
    mu <- sapply(mu_names, function(name) samples[[name]][c, s])

    sigma <- c(2, 0.5, 0.5)

    lambda_names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
    lambda <- sapply(lambda_names, function(name) samples[[name]][c, s])

    ys <- rep(0, length(xs))
    for (k in 1:3) {
      ys <- ys + lambda[k] * dnorm(xs, mu[k], sigma[k])
    }
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
}
```

In the first Markov chain the inferred behaviors of the first two components are swapped relative to the inferences in the second and third Markov chains. Because the component scales are fixed this exchange results in a slightly different mixture density function, but similar enough for both to be somewhat consistent with the observed data.

At the same time the behavior of the second and third components are swapped between the second and third Markov chains. In this case the known scales are the same and the resulting mixture probability density functions are identical.

```
xs <- seq(-12, 6, 0.25)

par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in c(1, 2, 3)) {
```

```
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```



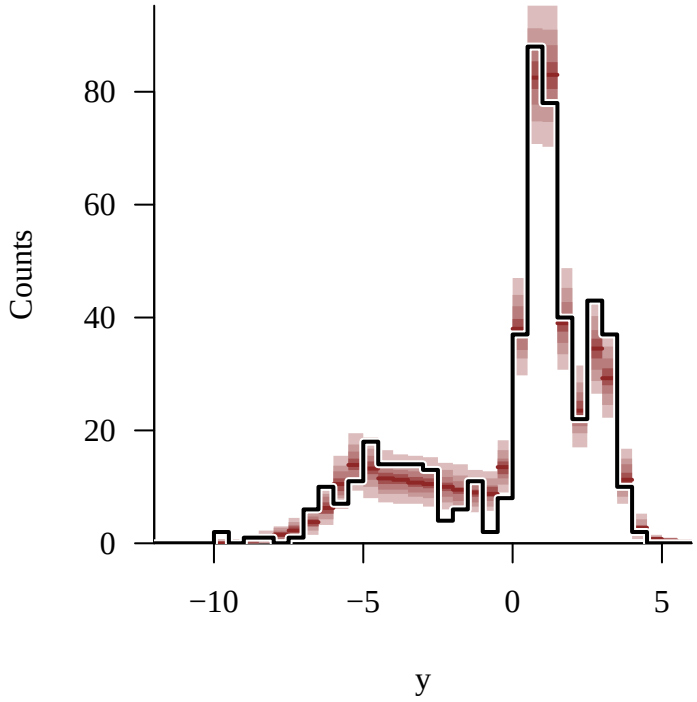
Given the multi-modality the contributions from the individual modes are almost surely not being weighted correctly. Consequently we cannot take our posterior quantification, and the resulting posterior predictive quantification, too seriously. That said, let's see what have.

Overall the retrodictive performance show no signs of problems.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 8 predictive values (0.0%) fell below the binning.
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 1147 predictive values (0.1%) fell above the binning.
```



Because of the multi-modality, however, it's a bit more fair to look at the retrodictive performance within individual Markov chain. Here we see that the retrodictive performance from the first Markov chains is similar to, but not exactly the same as, that from the second and third Markov chains, consistent with the inferred component behaviors. Moreover the retrodictive performance in the second and third Markov chains is the same, consistent with

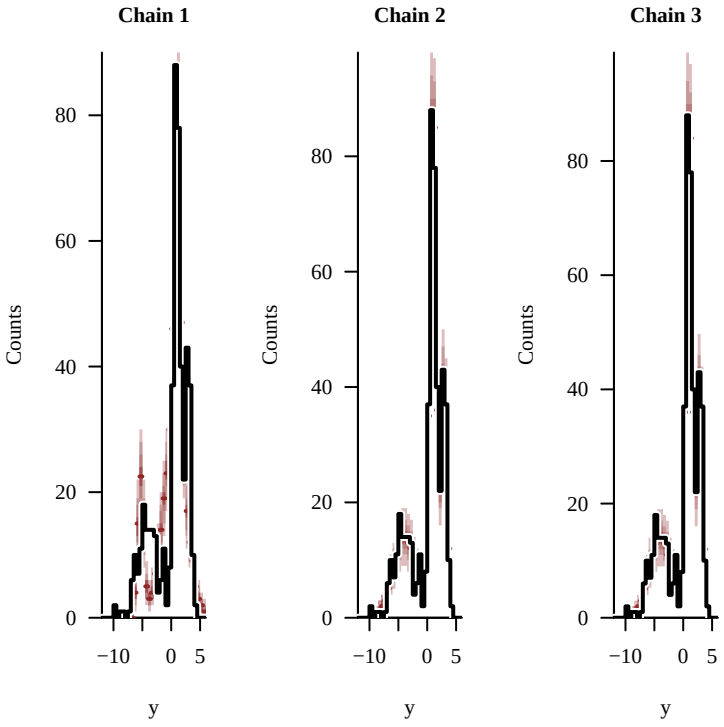
the corresponding modes capturing model configurations that are exact permutations of each other.

```
par(mfrow=c(1, 3), mar=c(5, 5, 3, 1))

for (c in 1:3) {
  ss <- lapply(samples, function(s) array(s[c,], dim=c(1, 1024)))
  util$plot_hist_quantiles(ss, 'y_pred', -12, 6, 0.5,
                           baseline_values=data$y, xlab="y",
                           main=paste0('Chain ', c))
}
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 1147 predictive values (0.2%) fell above the binning.
```

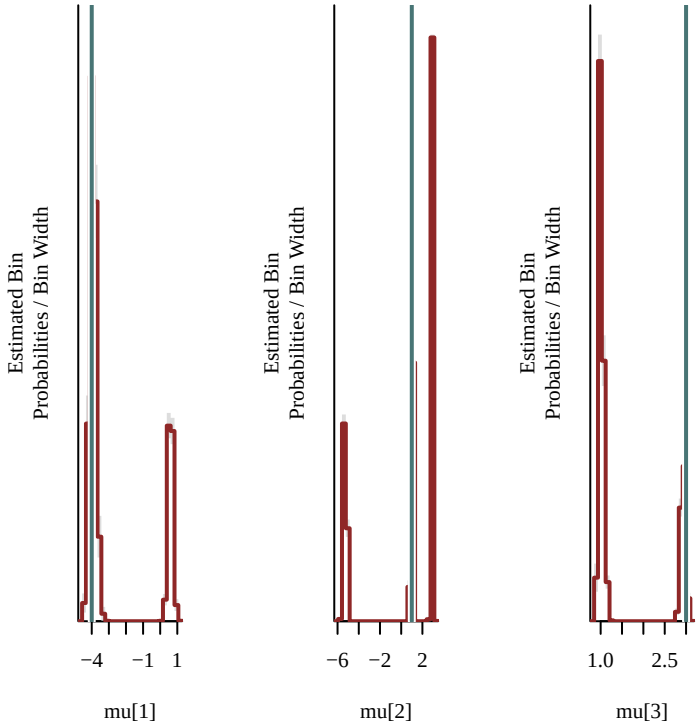
```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 3 predictive values (0.0%) fell below the binning.
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 4 predictive values (0.0%) fell below the binning.
```

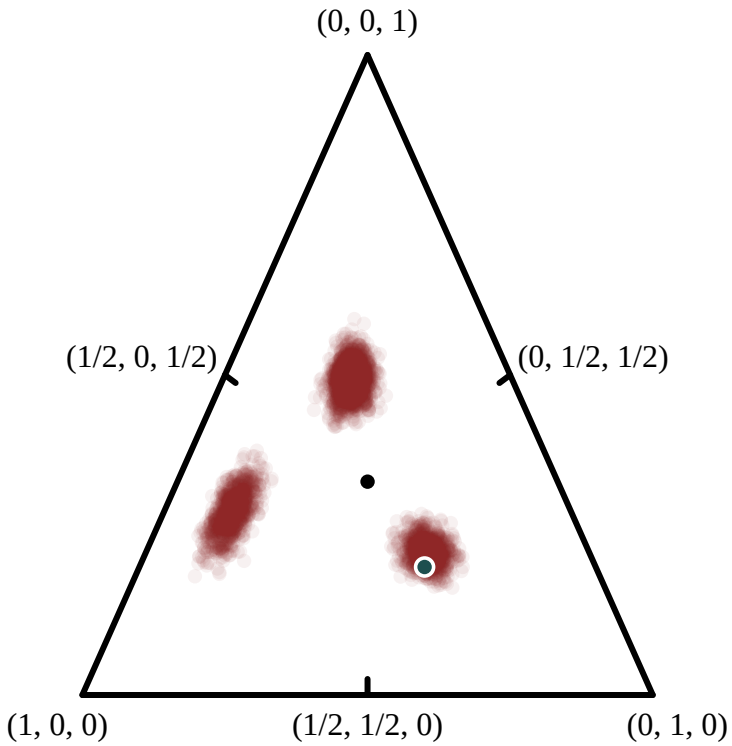


One of the posterior modes does appear to have captured the true model configuration, although again because we cannot rely on the relative weights of those modes we can't be certain how much the exact posterior distribution prefers that mode over the others.

```
par(mfrow=c(1, 3), mar=c(5, 5, 1, 1))

mu_true <- c(-4, 1, 3)
for (k in 1:3) {
  mu_name <- paste0('mu[', k, ']')
  util$plot_expectand_pushforward(samples[[mu_name]], 25,
                                  display_name=mu_name,
                                  baseline=mu_true[k],
                                  baseline_col=util$c_mid_teal)
}
```




```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

plot_simplex_samples(c(samples[['lambda[1]']], recursive=TRUE),
                     c(samples[['lambda[2]']], recursive=TRUE),
                     c(samples[['lambda[3]']], recursive=TRUE),
                     baseline=lambda_true)
```



Even if the observational model is exchangeable we can prevent the full Bayesian model from being exchangeable with an asymmetric prior model. Here let's assume that we have domain expertise that constrains the location of the component models to non-overlapping intervals,

$$
\begin{aligned}
-6 \lessapprox &\mu_1 \lessapprox -2 \\
-2 \lessapprox &\mu_1 \lessapprox +2 \\
+2 \lessapprox &\mu_1 \lessapprox +6.
\end{aligned}
$$

```
fit <- stan(file="stan_programs/normal_mix2b.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Despite this more informative prior model the parameters of the second and third component models still exhibit split $\hat{R}$ warnings.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

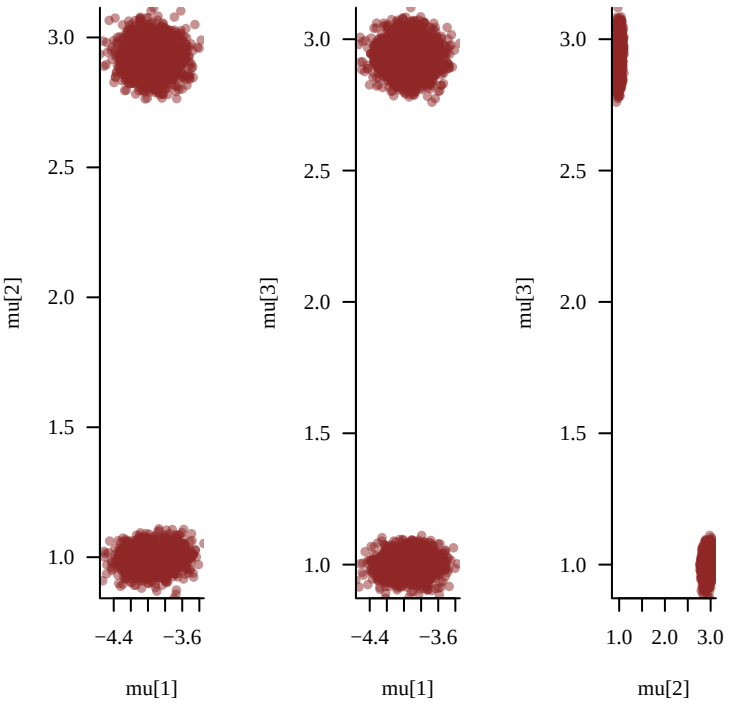
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
mu[2]:
  Split hat{R} (21.965) exceeds 1.1.

mu[3]:
  Split hat{R} (22.389) exceeds 1.1.

lambda[2]:
  Split hat{R} (6.540) exceeds 1.1.

lambda[3]:
  Split hat{R} (6.319) exceeds 1.1.


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

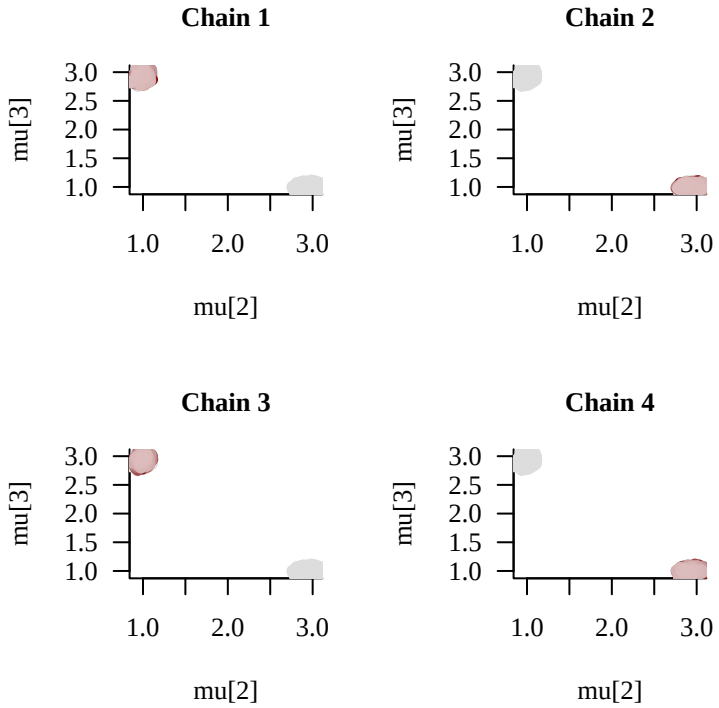
Indeed the multi-modality persists.

```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



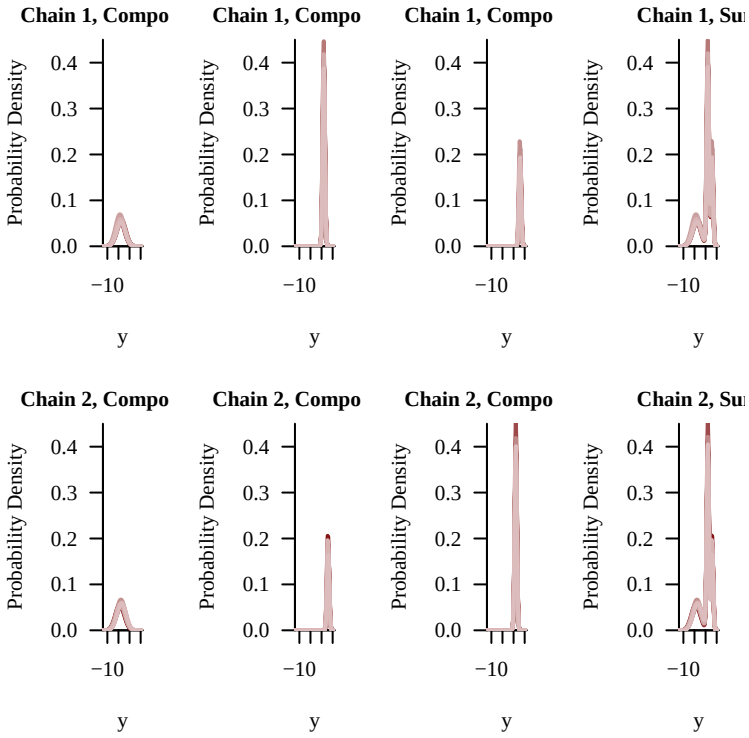
In fact one of the modes concentrates on model configurations with $\mu_3 < \mu_2$, directly contrasting with the structure of the prior model!

```
util$plot_pairs_by_chain(samples[['mu[2]']], 'mu[2]',
                         samples[['mu[3]']], 'mu[3]')
```



These two modes actually contain model configurations that permute the second and third component models entirely.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in 1:2) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```



The problem here is that, while strong prior models can suppress redundant or otherwise unwanted modes, they cannot eliminate them entirely. Consequently any modes in the likelihood function will persist into the posterior distribution, trapping Markov chains that get too close.

One way that we can eliminate the undesired modes is to remove entire regions of the model configuration space. For a mixture of one-dimensional normal models we can impose an ordering constraint on the component locations that completely removes all but one permutation of the component models. Effectively the ordering eliminates the redundancy without limiting the desired flexibility of the model.

```
fit <- stan(file="stan_programs/normal_mix2c.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Encouragingly all of the diagnostic warnings have ceased.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

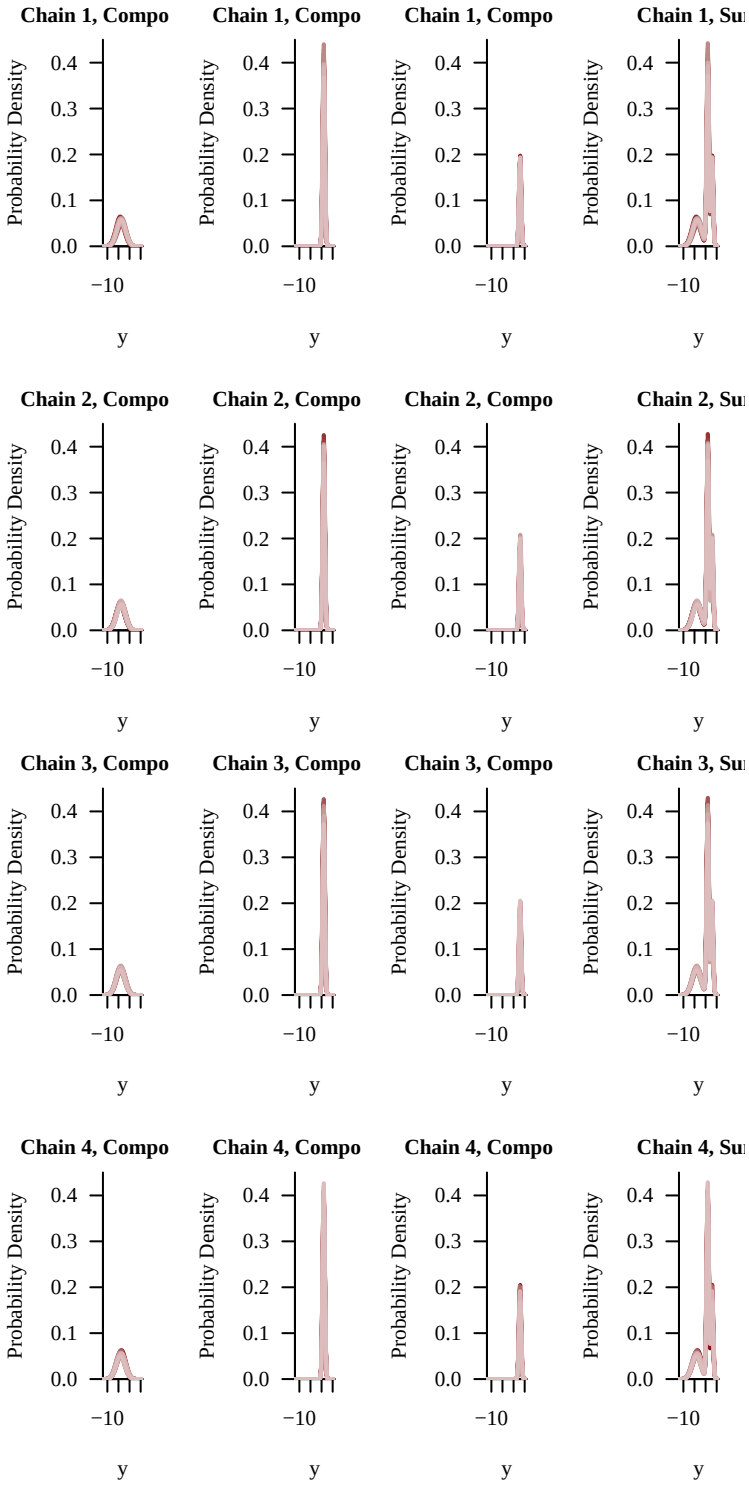
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

The inferred behavior is consistent across all four Markov chains.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in c(1, 2, 3, 4)) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```

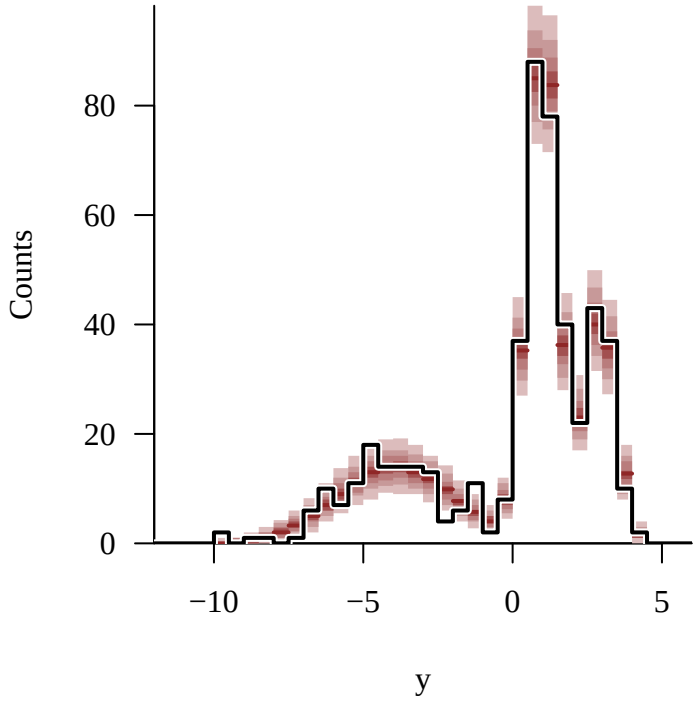


Now that our posterior computation is better behaved we can trust our posterior quantification.

First we check for any retrodictive tension.

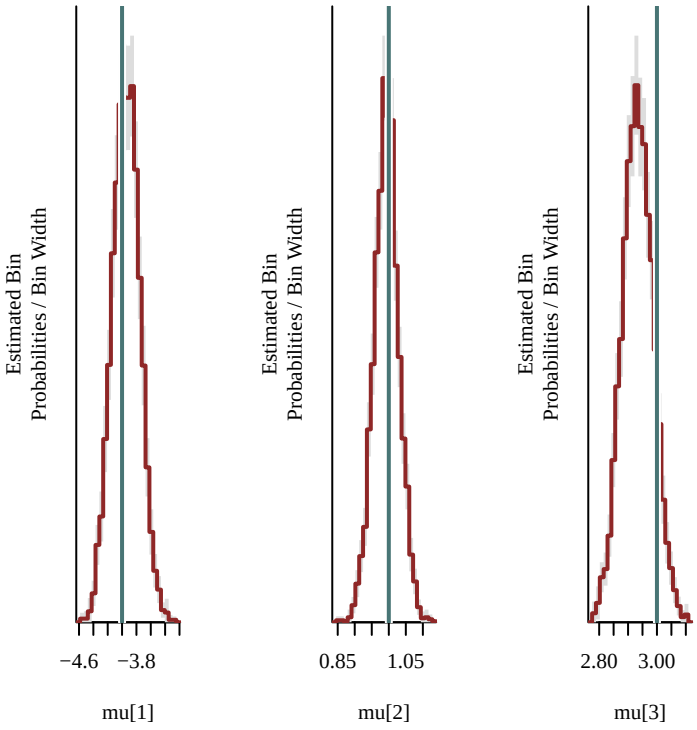
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 13 predictive values (0.0%) fell below the binning.
```

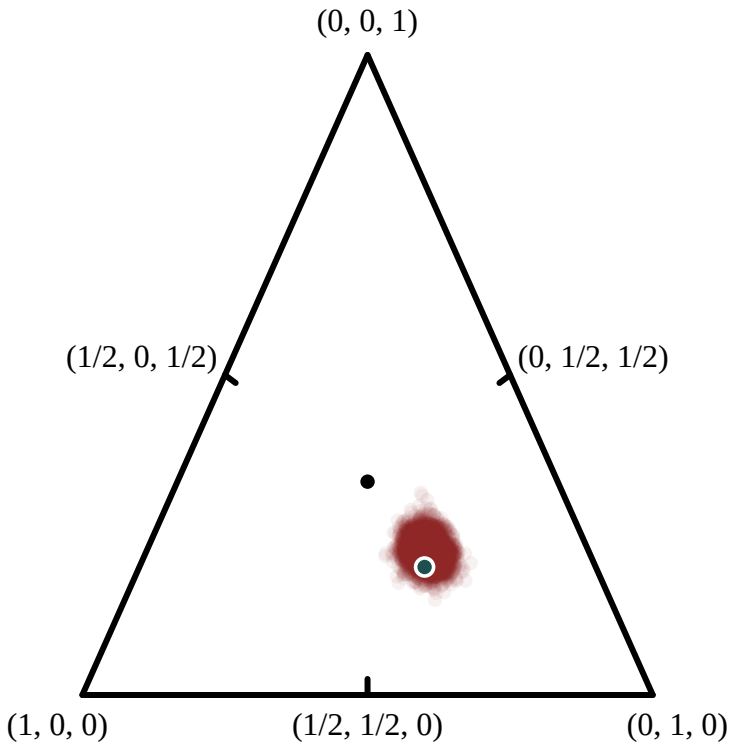


With no signs of inadequacies we can analyze our posterior inferences. Conveniently they all concentrate around the true behavior of the simulation data generating process.

```
par(mfrow=c(1, 3), mar=c(5, 5, 1, 1))

for (k in 1:3) {
  mu_name <- paste0('mu[', k, ']')
  util$plot_expectand_pushforward(samples[[mu_name]], 25,
                                  display_name=mu_name,
                                  baseline=mu_true[k],
                                  baseline_col=util$c_mid_teal)
}
```




```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

plot_simplex_samples(c(samples[['lambda[1]']], recursive=TRUE),
                     c(samples[['lambda[2]']], recursive=TRUE),
                     c(samples[['lambda[3]']], recursive=TRUE),
                     baseline=lambda_true)
```



4.5.3 Unknown Component Probabilities, Locations, and Scales

Why don't we push the analysis even further and try to infer *all* of the component model configurations at the same time. In this case nothing distinguishes the component models from each other. The model has reached peak exchangeability.

```
fit <- stan(file="stan_programs/normal_mix3a.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The computational diagnostics are now so generous that all of the parameters get a split $\hat{R}$ warning.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

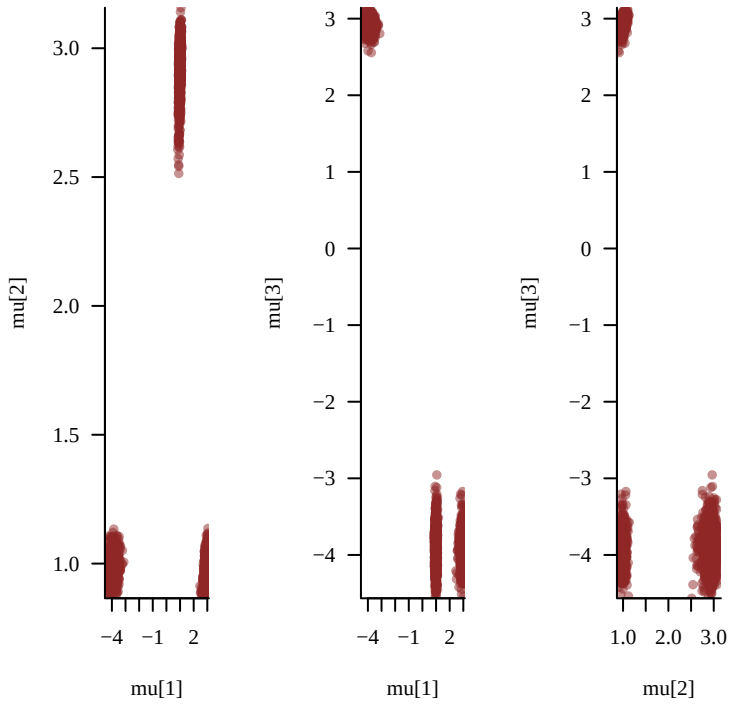
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'sigma', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

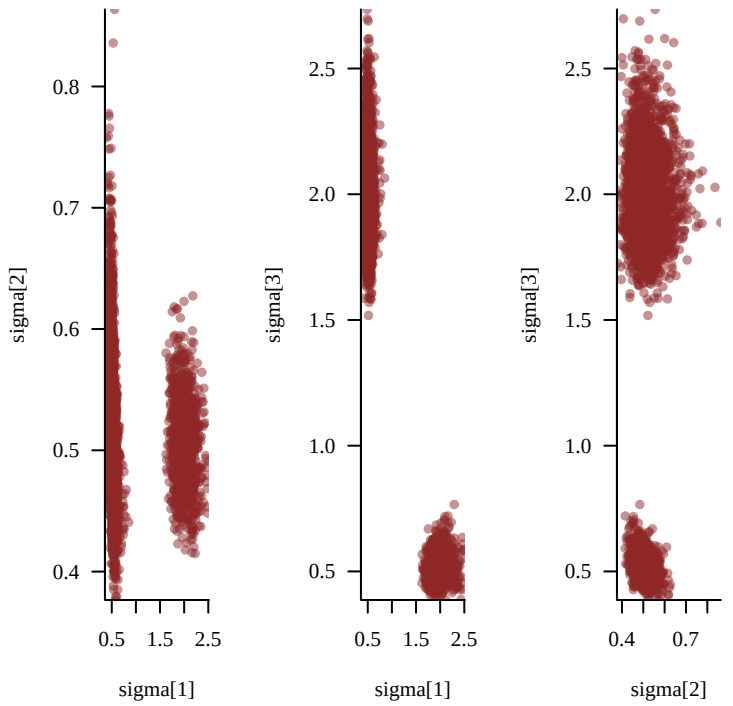
```
mu[1]:
  Split hat{R} (22.613) exceeds 1.1.

mu[2]:
  Split hat{R} (15.986) exceeds 1.1.

mu[3]:
  Split hat{R} (17.349) exceeds 1.1.

sigma[1]:
  Split hat{R} (7.667) exceeds 1.1.

sigma[3]:
  Split hat{R} (4.759) exceeds 1.1.

lambda[1]:
  Split hat{R} (5.073) exceeds 1.1.

lambda[2]:
  Split hat{R} (5.524) exceeds 1.1.

lambda[3]:
  Split hat{R} (1.561) exceeds 1.1.


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

Examining some pairs plots we can see the horde of modes responsible for the split $\hat{R}$ warnings.

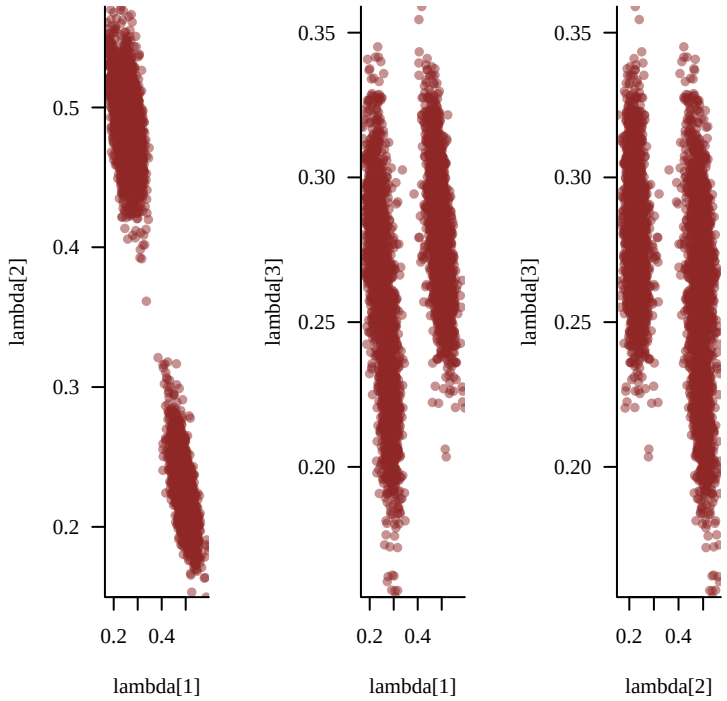
```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



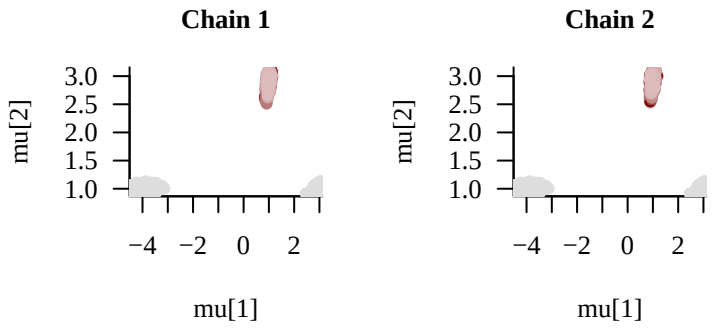

```
names <- sapply(1:3, function(k) paste0('sigma[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```

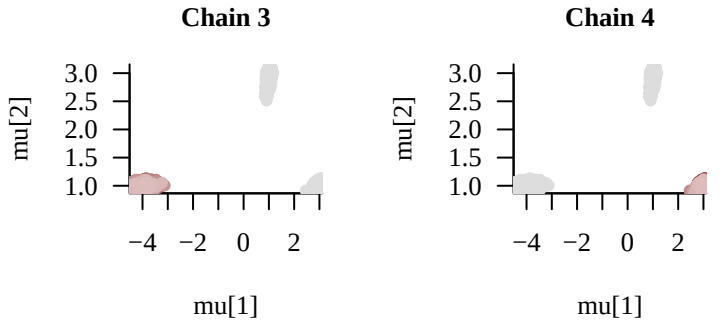


```
names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```

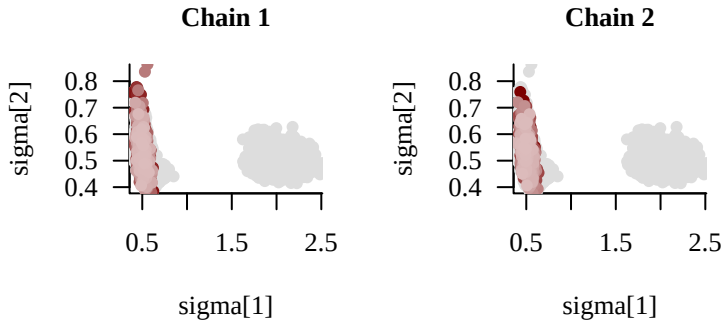


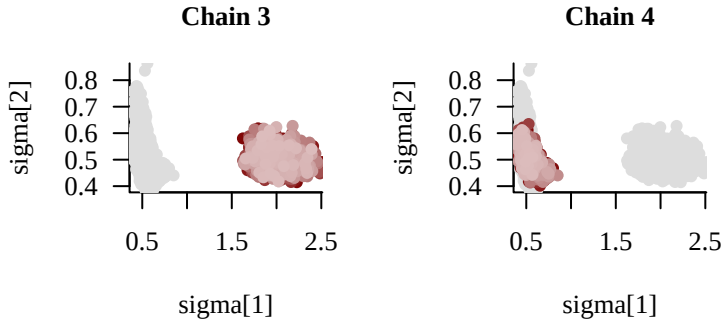

```
util$plot_pairs_by_chain(samples[['mu[1]']], 'mu[1]',
                         samples[['mu[2]']], 'mu[2]')
```



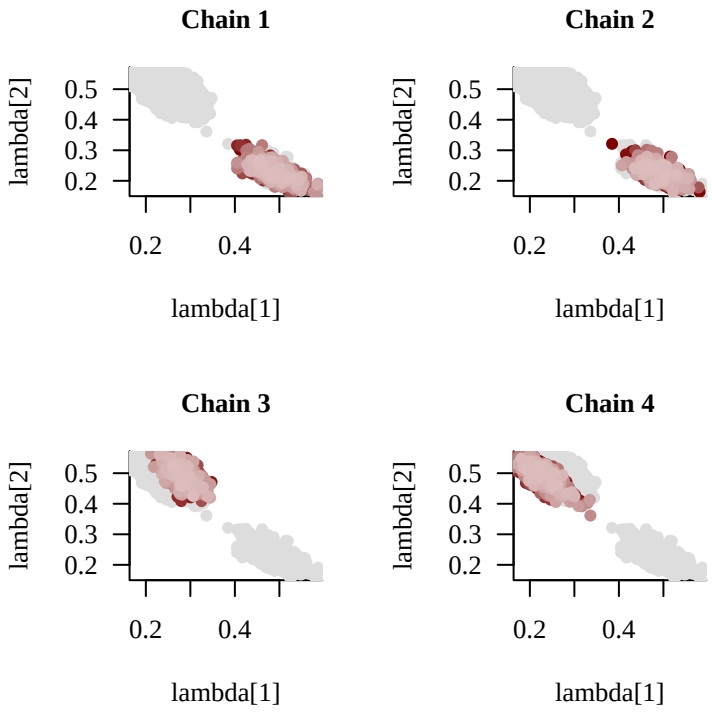



```
util$plot_pairs_by_chain(samples[['sigma[1]']], 'sigma[1]',
                         samples[['sigma[2]']], 'sigma[2]')
```





```
util$plot_pairs_by_chain(samples[['lambda[1]']], 'lambda[1]',
                         samples[['lambda[2]']], 'lambda[2]')
```

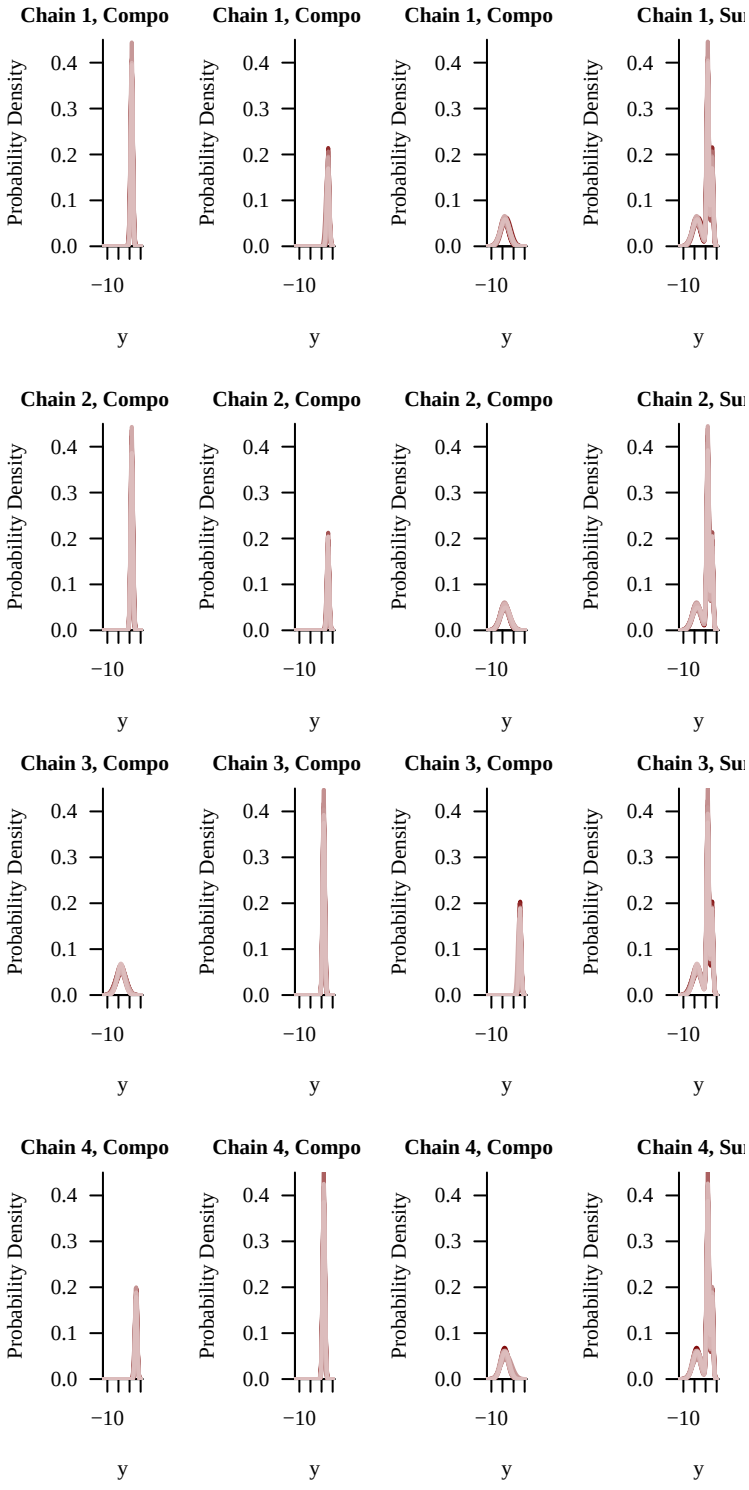


Taking a closer look at the inferred behavior of the component models we can see that all of these modes appear to be permutations of each other.

```
plot_component_realizations <- function(k, c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_name <- paste0('mu[', k, ']')
    mu <- samples[[mu_name]][c, s]

    sigma_name <- paste0('sigma[', k, ']')
    sigma <- samples[[sigma_name]][c, s]

    lambda_name <- paste0('lambda[', k, ']')
    lambda <- samples[[lambda_name]][c, s]

    ys <- lambda * dnorm(xs, mu, sigma)
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
```

```
}

plot_sum_realizations <- function(c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
    mu <- sapply(mu_names, function(name) samples[[name]][c, s])

    sigma_names <- sapply(1:3, function(k) paste0('sigma[', k, ']'))
    sigma <- sapply(sigma_names, function(name) samples[[name]][c, s])

    lambda_names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
    lambda <- sapply(lambda_names, function(name) samples[[name]][c, s])

    ys <- rep(0, length(xs))
    for (k in 1:3) {
      ys <- ys + lambda[k] * dnorm(xs, mu[k], sigma[k])
    }
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
}
```

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in 1:4) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```



Can breaking the exchangeability with an ordering on the component locations resolve some

of this degeneracy?

```
fit <- stan(file="stan_programs/normal_mix3b.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Well we don't see quite as many split $\hat{R}$ warnings, so perhaps we've made some progress.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

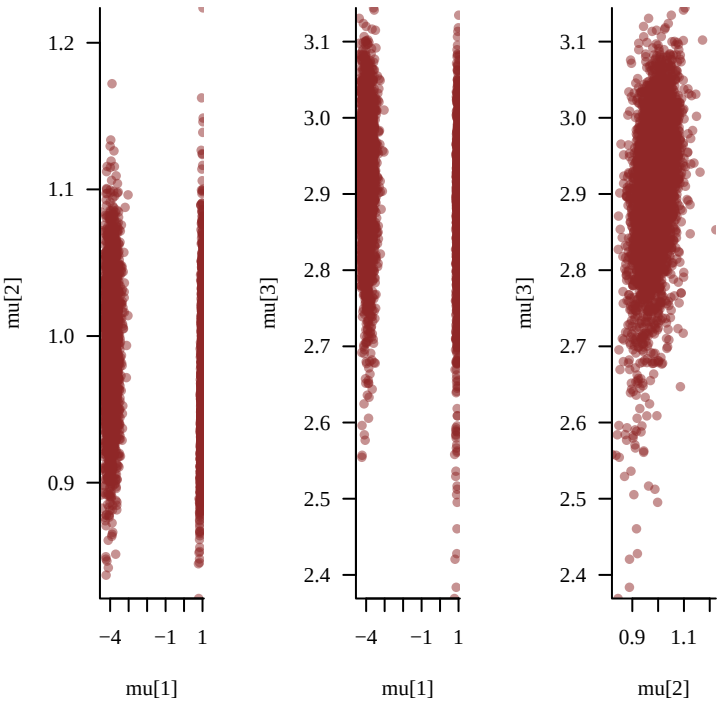
```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'sigma', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
mu[1]:
  Split hat{R} (12.694) exceeds 1.1.

sigma[1]:
  Split hat{R} (5.107) exceeds 1.1.

sigma[2]:
  Split hat{R} (14.117) exceeds 1.1.

sigma[3]:
  Split hat{R} (1.119) exceeds 1.1.

lambda[1]:
  Split hat{R} (2.603) exceeds 1.1.

lambda[2]:
  Split hat{R} (1.310) exceeds 1.1.

lambda[3]:
  Split hat{R} (1.701) exceeds 1.1.
```

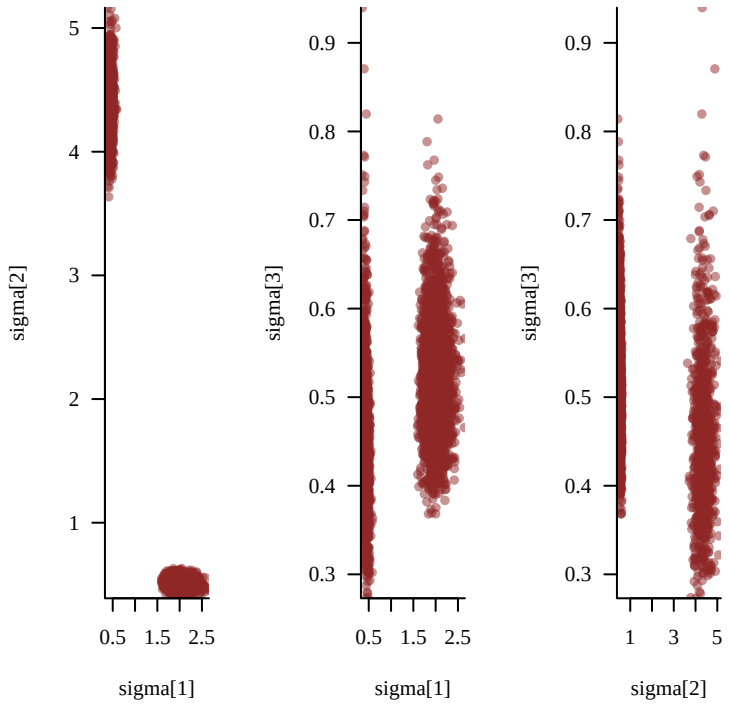
```
Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

It looks like we may be down to just two modes.

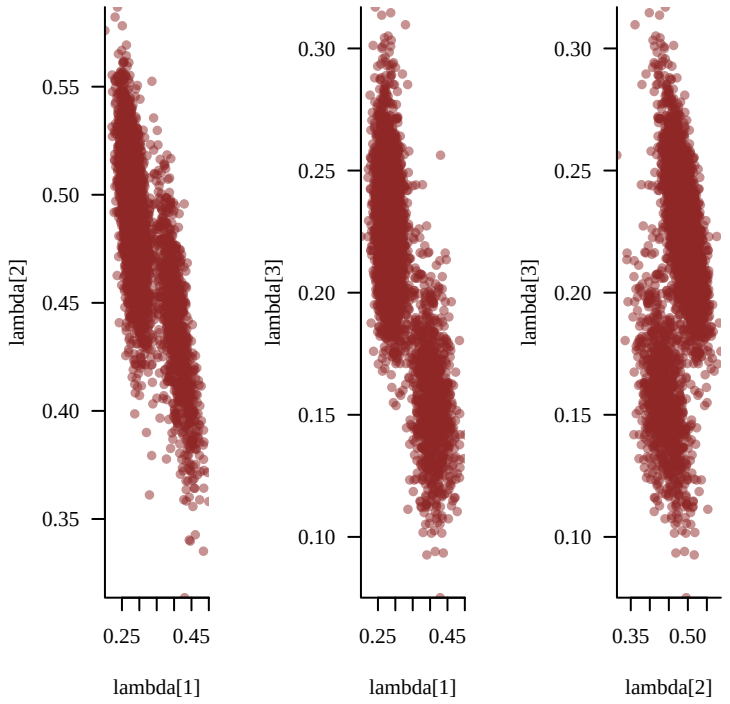
```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```




```
names <- sapply(1:3, function(k) paste0('sigma[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```

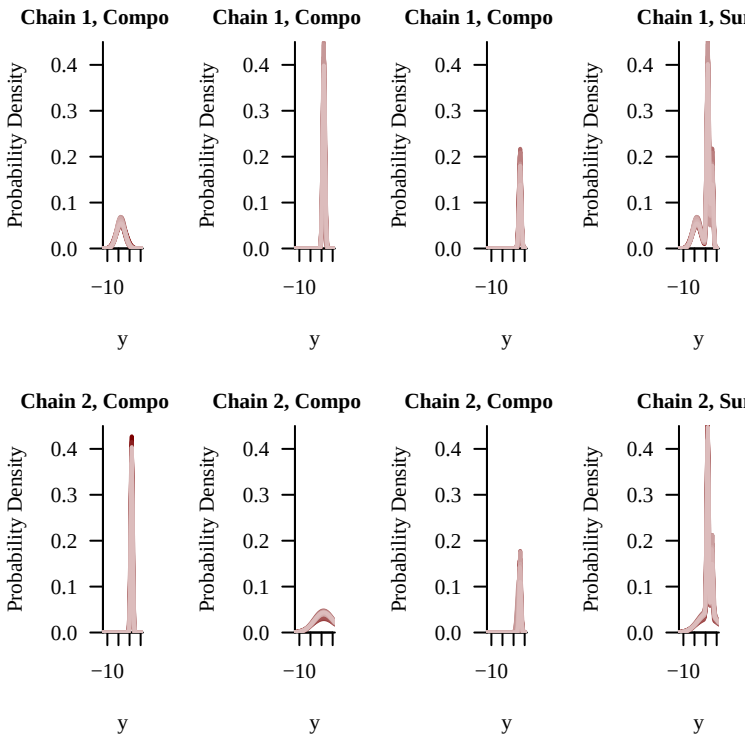


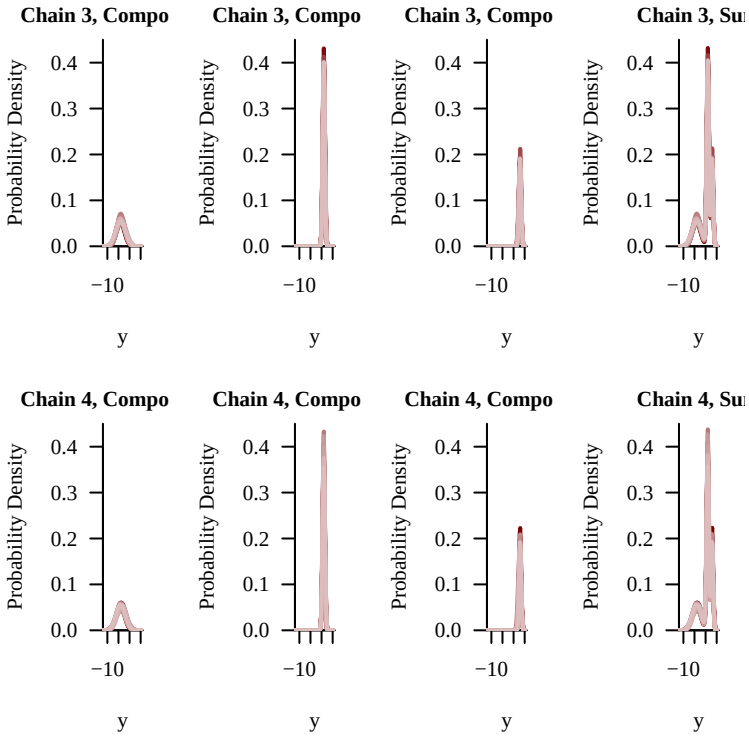

```
names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



Investigating the inferred component behaviors more closely reveals something interesting.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in 1:4) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```





The first, third, and fourth Markov chains exhibit the ideal behavior, with each component matching the behavior of the true component data generating processes. In the second Markov chain, however, the first component model moves up to capture the second true component data generating process. Because of the ordering constraint this pushes the second component model up. The second component model then uses the pliability of the component scales to expand and cover the observed data at smaller values that the first component missed.

This latter contortion is less consistent with the observed data, which we can see in the corresponding retrodictive checks.

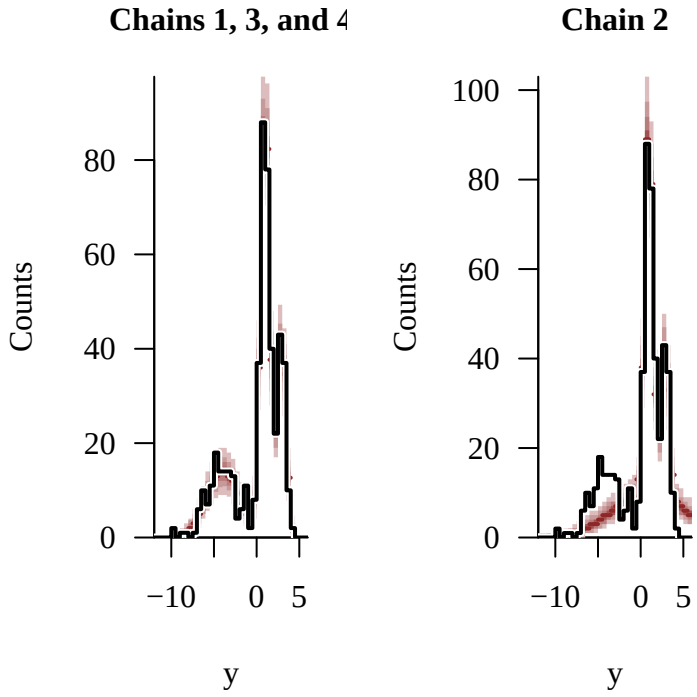
```
par(mfrow=c(1, 2), mar=c(5, 5, 3, 1))

samples134 <- lapply(samples, function(s) s[c(1, 3, 4),])
util$plot_hist_quantiles(samples134, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y",
                         main="Chains 1, 3, and 4")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 16 predictive values (0.0%) fell below the binning.
```

```
samples2 <- lapply(samples, function(s) array(s[2,], dim=c(1, 1024)))
util$plot_hist_quantiles(samples2, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y",
                         main="Chain 2")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 354 predictive values (0.1%) fell below the binning.
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 27805 predictive values (5.4%) fell above the binning.
```



The exact posterior distribution will allocate less probability to the second mode, allowing the behaviors in the first mode to dominate our inferences. Unfortunately Markov chain Monte Carlo cannot reliably estimate the relative probabilities of the two modes, and in practice we don't know how much to discount the second mode.

4.5.4 Unknown Number of Components

Finally let's see what happens when we use a mixture model with more components than are actually in the simulation data generating process. Note that we're starting with an ordering constraint to make the posterior distribution as well-behaved as possible.

```
data$K <- 5

fit <- stan(file="stan_programs/normal_mix4.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

We have good news and we have bad news. The good news is that there are almost no split $\hat{R}$ warnings; the bad news is that just about every other warning shows up. In particular the tree depth saturation and small empirical effective sample size warnings suggest strong Markov chain autocorrelations which themselves hint at complex posterior uncertainties.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  Chain 1: 1 of 1024 transitions (0.1%) diverged.
  Chain 1: 52 of 1024 transitions (5.078125%) saturated the maximum treedepth of 10.

  Chain 2: 36 of 1024 transitions (3.515625%) saturated the maximum treedepth of 10.

  Chain 3: 116 of 1024 transitions (11.328125%) saturated the maximum treedepth of 10.

  Chain 4: 182 of 1024 transitions (17.7734375%) saturated the maximum treedepth of 10.

  Divergent Hamiltonian transitions result from unstable numerical
trajectories.  These instabilities are often due to degenerate target
geometry, especially "pinches".  If there are only a small number of
divergences then running with adept_delta larger than 0.801 may reduce
the instabilities at the cost of more expensive Hamiltonian
transitions.

  Numerical trajectories that saturate the maximum treedepth have
terminated prematurely.  Increasing max_depth above 10 should result in
more expensive, but more efficient, Hamiltonian transitions.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'sigma', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

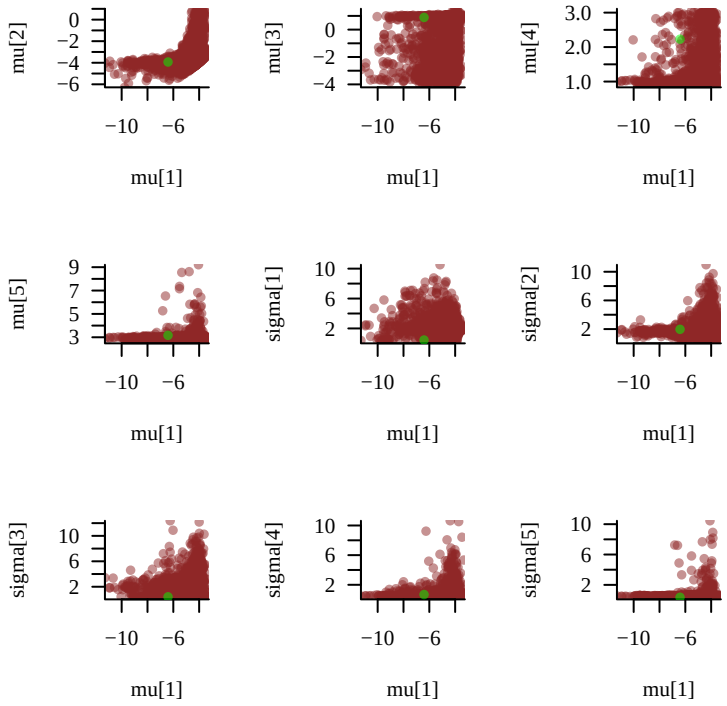
```
mu[1]:
```

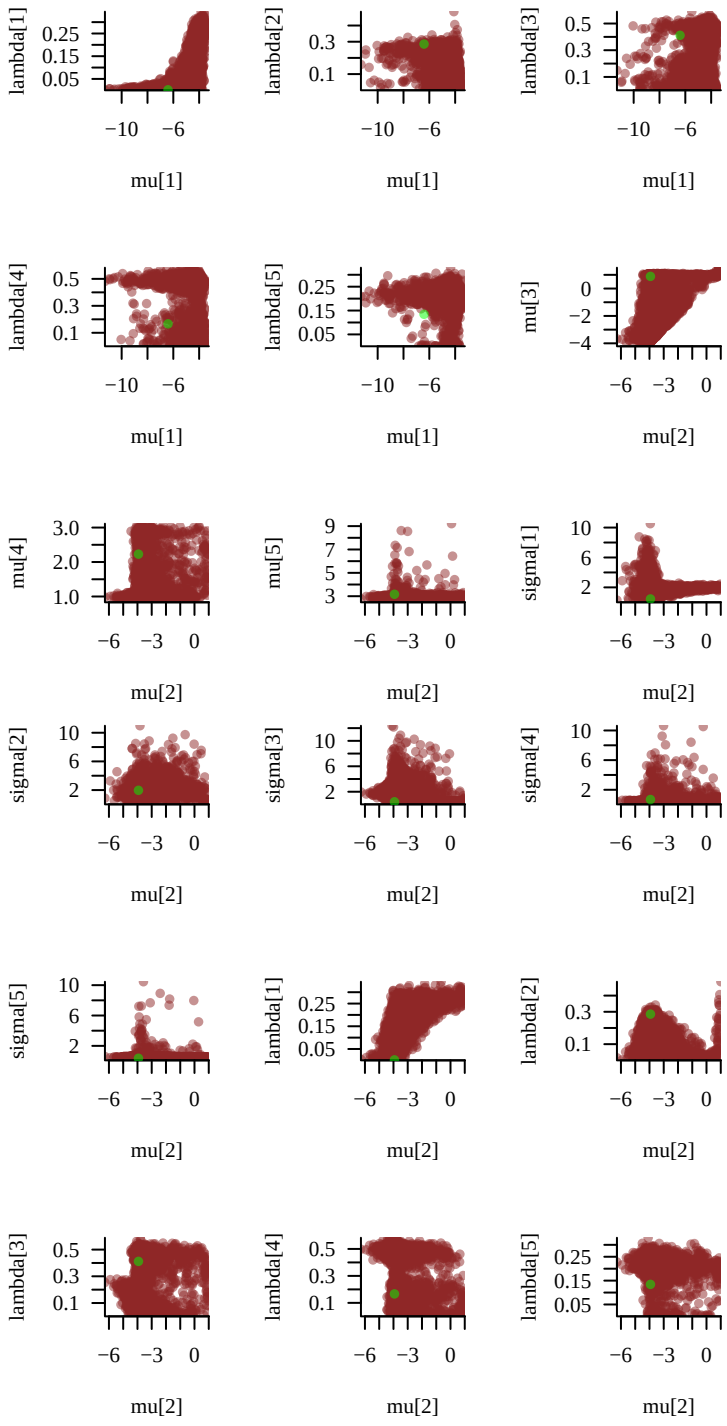
```
  Chain 1: Left tail hat{xi} (0.409) exceeds 0.25.
  Chain 3: Left tail hat{xi} (0.320) exceeds 0.25.
  Chain 2: hat{ESS} (92.084) is smaller than desired (100).
  Chain 3: hat{ESS} (86.435) is smaller than desired (100).

mu[2]:
  Chain 3: hat{ESS} (84.762) is smaller than desired (100).

mu[3]:
  Chain 1: hat{ESS} (30.790) is smaller than desired (100).
  Chain 2: hat{ESS} (47.728) is smaller than desired (100).
  Chain 3: hat{ESS} (34.329) is smaller than desired (100).
  Chain 4: hat{ESS} (30.072) is smaller than desired (100).

mu[4]:
  Chain 2: Right tail hat{xi} (0.484) exceeds 0.25.
  Chain 3: Right tail hat{xi} (1.429) exceeds 0.25.
  Chain 1: hat{ESS} (27.742) is smaller than desired (100).
  Chain 2: hat{ESS} (32.215) is smaller than desired (100).
  Chain 3: hat{ESS} (29.548) is smaller than desired (100).
  Chain 4: hat{ESS} (26.779) is smaller than desired (100).

mu[5]:
  Chain 1: Right tail hat{xi} (0.797) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.383) exceeds 0.25.
  Chain 4: hat{ESS} (87.885) is smaller than desired (100).

sigma[1]:
  Chain 1: Right tail hat{xi} (0.281) exceeds 0.25.
  Chain 2: Right tail hat{xi} (0.339) exceeds 0.25.
  Chain 3: Right tail hat{xi} (0.311) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.329) exceeds 0.25.

sigma[3]:
  Chain 1: hat{ESS} (58.160) is smaller than desired (100).
  Chain 3: hat{ESS} (52.130) is smaller than desired (100).
  Chain 4: hat{ESS} (40.478) is smaller than desired (100).

sigma[4]:
  Chain 1: Right tail hat{xi} (0.626) exceeds 0.25.
  Chain 2: Right tail hat{xi} (0.602) exceeds 0.25.
  Chain 3: Right tail hat{xi} (1.218) exceeds 0.25.
  Chain 4: Right tail hat{xi} (1.221) exceeds 0.25.
```

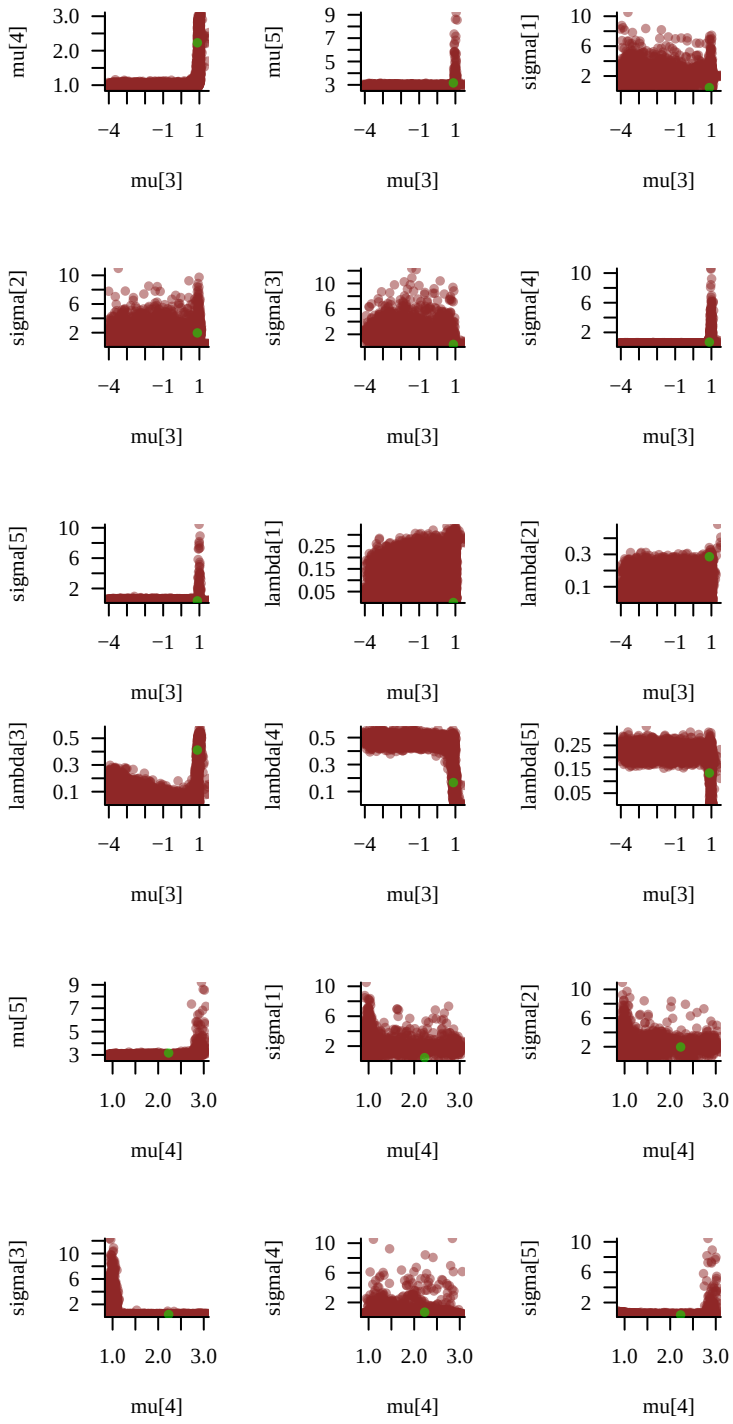
```
  Chain 2: hat{ESS} (42.709) is smaller than desired (100).
  Chain 3: hat{ESS} (52.978) is smaller than desired (100).

sigma[5]:
  Chain 1: Right tail hat{xi} (0.947) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.632) exceeds 0.25.
  Chain 4: hat{ESS} (90.384) is smaller than desired (100).

lambda[1]:
  Chain 3: hat{ESS} (94.205) is smaller than desired (100).

lambda[3]:
  Split hat{R} (1.109) exceeds 1.1.
  Chain 1: hat{ESS} (24.711) is smaller than desired (100).
  Chain 2: hat{ESS} (31.587) is smaller than desired (100).
  Chain 3: hat{ESS} (26.128) is smaller than desired (100).
  Chain 4: hat{ESS} (22.753) is smaller than desired (100).

lambda[4]:
  Chain 2: Left tail hat{xi} (0.675) exceeds 0.25.
  Chain 3: Left tail hat{xi} (0.914) exceeds 0.25.
  Split hat{R} (1.113) exceeds 1.1.
  Chain 1: hat{ESS} (27.508) is smaller than desired (100).
  Chain 2: hat{ESS} (21.237) is smaller than desired (100).
  Chain 3: hat{ESS} (23.639) is smaller than desired (100).
  Chain 4: hat{ESS} (20.943) is smaller than desired (100).

lambda[5]:
  Chain 3: Left tail hat{xi} (0.269) exceeds 0.25.
  Chain 4: Left tail hat{xi} (0.337) exceeds 0.25.
  Chain 1: hat{ESS} (37.481) is smaller than desired (100).
  Chain 3: hat{ESS} (61.483) is smaller than desired (100).
  Chain 4: hat{ESS} (40.355) is smaller than desired (100).


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.

Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.

Small empirical effective sample sizes result in imprecise Markov chain
Monte Carlo estimators.
```

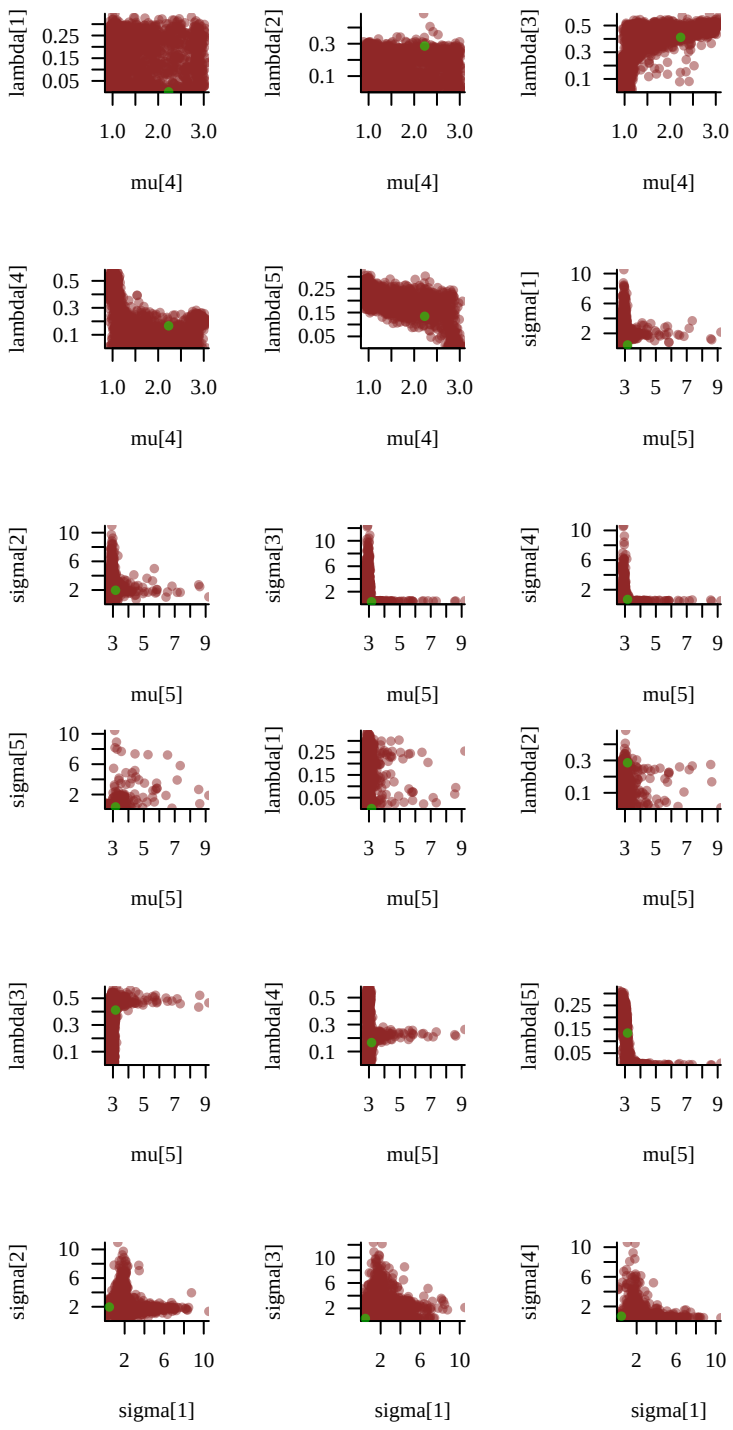
In general I do not recommend looking at every possible pairs plot; instead I prefer using the model structure to prioritize a reasonable number of pairs plots. In this case, however, I want to show all of the pairs plots just to demonstrate how degenerate the mixture model posterior distribution has become. It truly is the stuff of statistical nightmares.

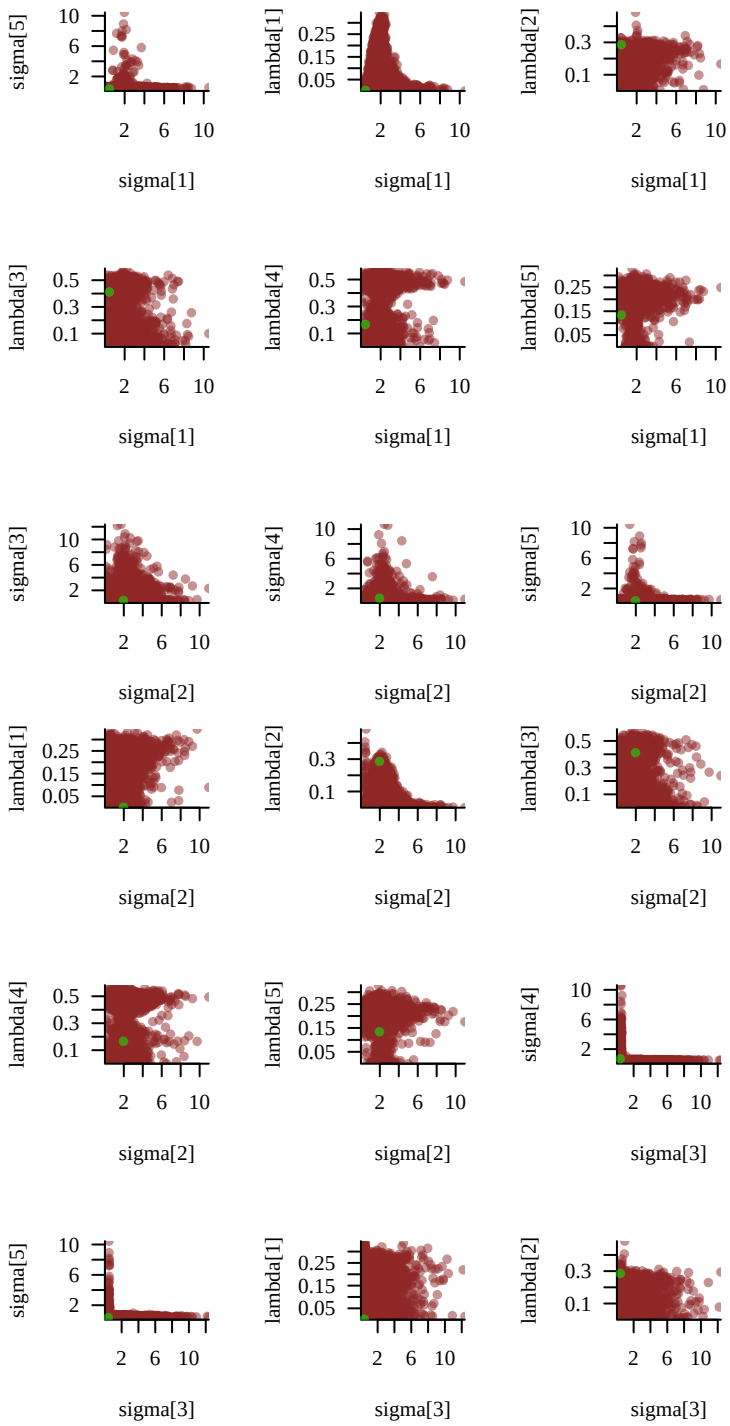
```
names <- c(sapply(c('mu', 'sigma', 'lambda'), function(name)
           sapply(1:5, function(k) paste0(name, '[', k, ']'))))
util$plot_div_pairs(names, names, samples, diagnostics)
```

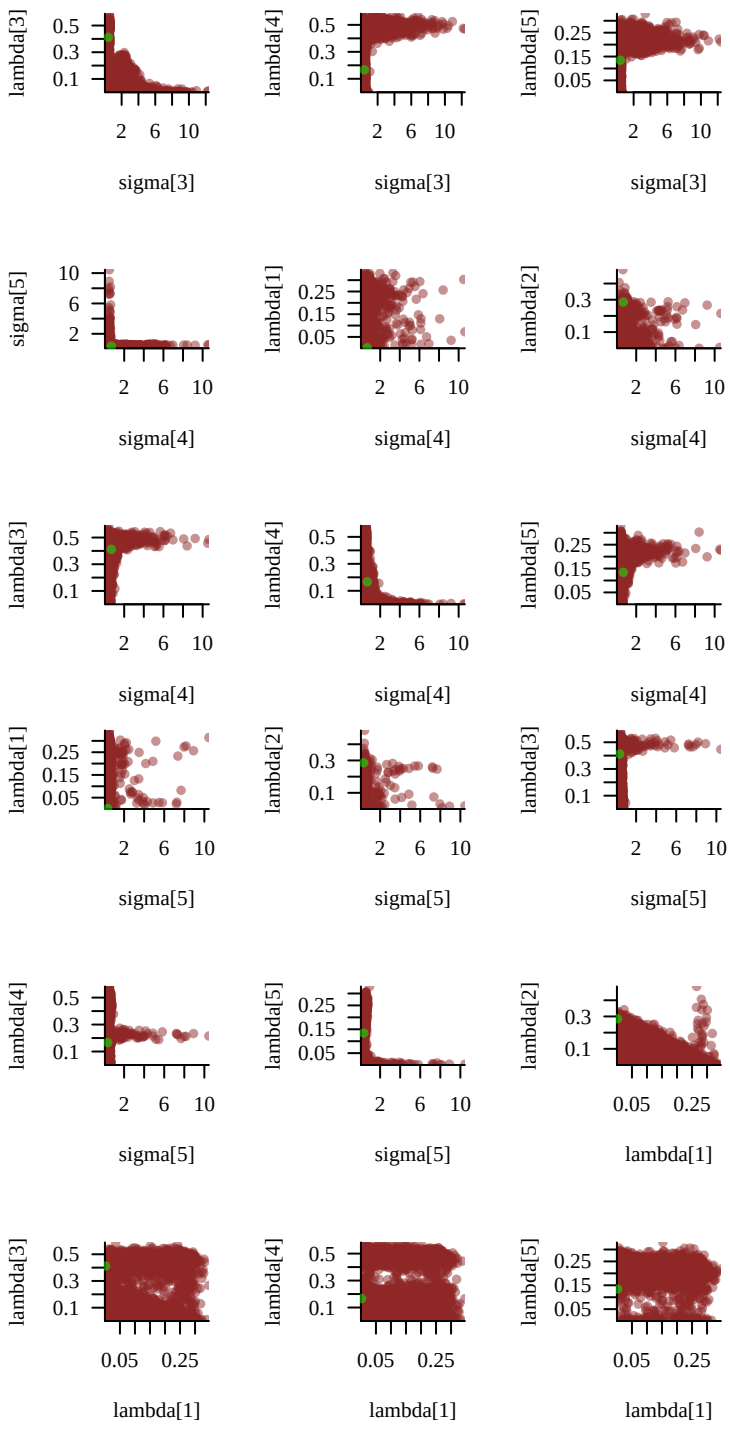


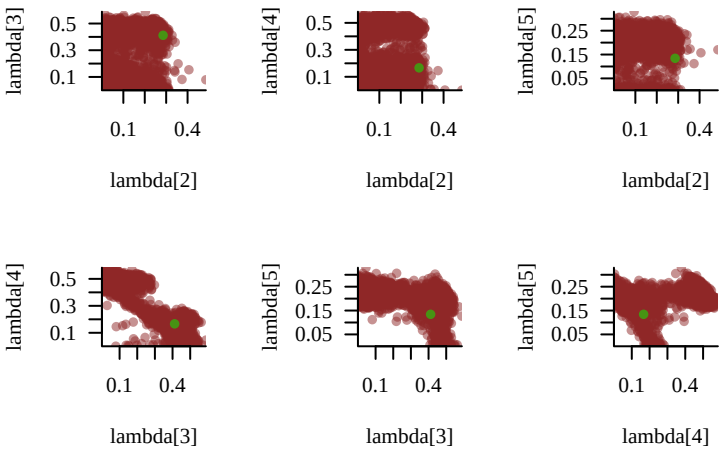

lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
mu[3]
mu[4]
mu[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
mu[1]
mu[2]


mu[4]
mu[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
mu[3]
mu[4]


lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
mu[4]
mu[5]
sigma[1]

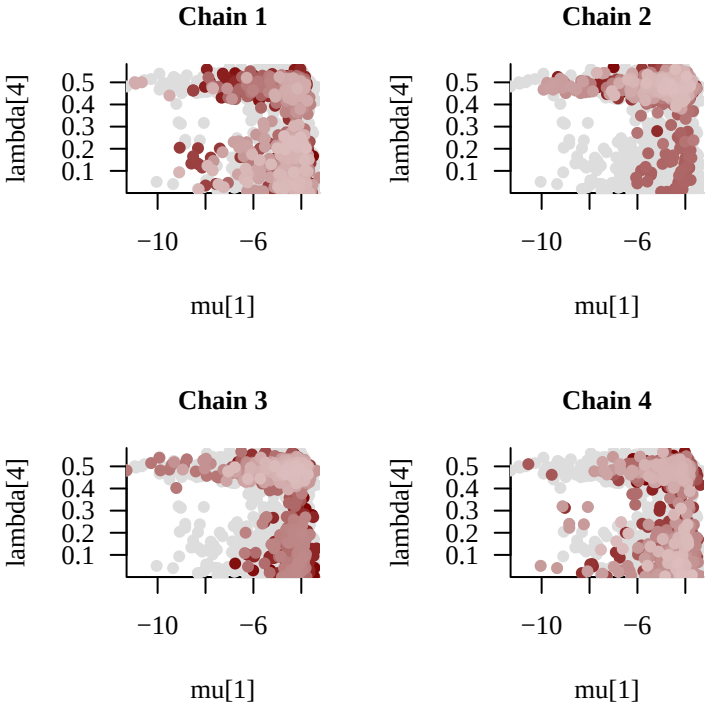

sigma[5]
lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
sigma[3]
sigma[4]
sigma[5]
lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
sigma[4]
sigma[5]
lambda[1]
lambda[2]
sigma[1]
sigma[2]
sigma[3]


lambda[3]
sigma[3]
lambda[4]
sigma[3]
lambda[5]
sigma[3]
sigma[5]
sigma[4]
lambda[1]
sigma[4]
lambda[2]
sigma[4]
lambda[3]
sigma[4]
lambda[4]
sigma[4]
lambda[5]
sigma[4]
lambda[1]
sigma[5]
lambda[2]
sigma[5]
lambda[3]
sigma[5]
lambda[4]
sigma[5]
lambda[5]
sigma[5]
lambda[2]
lambda[1]
lambda[3]
lambda[1]
lambda[4]
lambda[1]
lambda[5]
lambda[1]



Interestingly there don't appear to be any isolated modes. To be clear we do see some modal structure, for example in the plot of `mu[1]` against `lambda[4]`, but the individual Markov chains are largely able to transition between the modes without issue. Consequently our Markov chain Monte Carlo estimators are reasonably reliable.

```
util$plot_pairs_by_chain(samples[['mu[1]']], 'mu[1]',
                         samples[['lambda[4]']], 'lambda[4]')
```

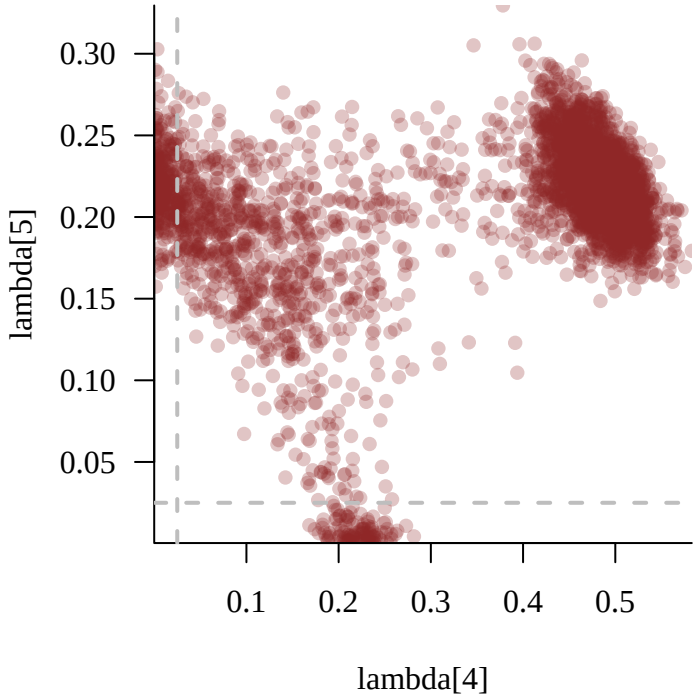


To better understand all of these intricate degeneracies let's take a look at the underlying mixture model and reason about the many different ways that it can contort itself while

maintaining consistency with the observed data. Ultimately a mixture model with too many components needs to find a way to hide, if not outright eliminate, the contribution from the extraneous components.

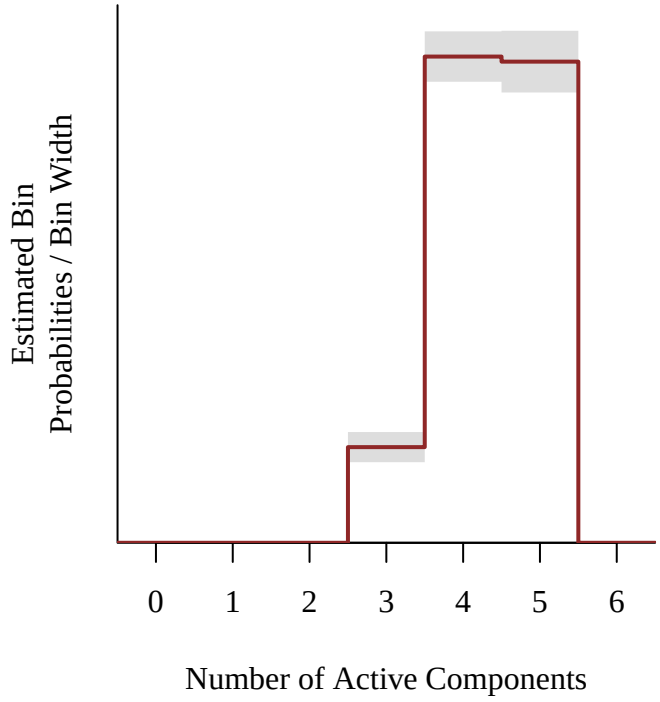
For example some of the contribution from some of the component models can be "turned off" if the corresponding component probabilities are close to zero. Indeed we can see bands of low component probabilities in the pairs plots.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

plot(c(samples[['lambda[4]']], recursive=TRUE),
     c(samples[['lambda[5]']], recursive=TRUE),
     pch=16, col="#8F272744",
     xlab='lambda[4]', ylab='lambda[5]')
abline(h=0.025, col='gray', lty=2, lwd=2)
abline(v=0.025, col='gray', lty=2, lwd=2)
```



Let's use the structure of these bands to define a cut-off between the "active" components that substantially contribute to the overall mixture probability distribution and the "inactive" components that offer only negligible contributions. This cut-off then allows us to approximately count the number of active components in each model configuration.

```
var_repl <- list('lambda'=array(sapply(1:5,
                                       function(k) paste0("lambda[", k, "]"))))
active_components <-
  util$eval_expectand_pushforward(samples,
                                  function(lambda) sum(lambda > 0.025),
                                  var_repl)

par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(active_components,
                                7, flim=c(-0.5, 6.5),
                                display_name="Number of Active Components")
```



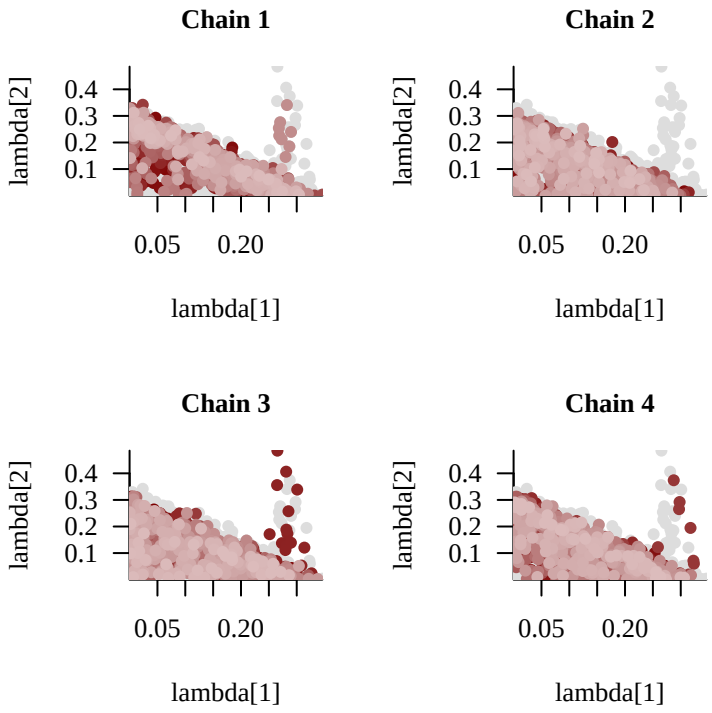
The posterior distribution concentrates on model configurations with three, four, or five active components. Does this behavior align with our understanding of the model and the simulation data generating process?

Well the suppression of model configurations with only one or two active components make sense. None of these model configurations are able to capture the three-peaks in the observed data.

The model configurations with three active components are perhaps the most intuitive given the three-component structure of the simulation data generating process. Interestingly *which*

two components are turned off is not fixed but rather vary as the Markov chains evolve. For example sometimes `lambda[1]` is close to zero while `lambda[2]` is not and sometimes `lambda[2]` is close to zero while `lambda[1]` is not. Sometimes they're both far enough away from zero for the corresponding component models to contribute to the mixture probability distribution, and sometimes they're both close enough to zero for the corresponding contributions to be negligible.

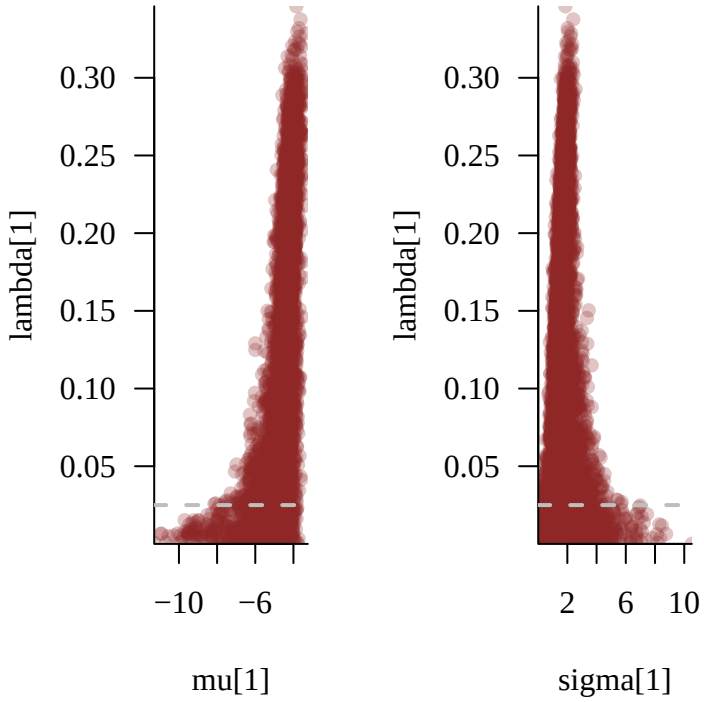
```
util$plot_pairs_by_chain(samples[['lambda[1]']], 'lambda[1]',
                         samples[['lambda[2]']], 'lambda[2]')
```



Note also that when a component becomes inactive the corresponding location and scale parameters are no longer informed by the observed data and instead relax back to the prior model. This explains some of the ridges that we see in the pairs plots.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

plot(c(samples[['mu[1]']], recursive=TRUE),
     c(samples[['lambda[1]']], recursive=TRUE),
     pch=16, col="#8F272744",
     xlab='mu[1]', ylab='lambda[1]')
abline(h=0.025, col='gray', lty=2, lwd=2)
```

```
plot(c(samples[['sigma[1]']], recursive=TRUE),
     c(samples[['lambda[1]']], recursive=TRUE),
     pch=16, col="#8F272744",
     xlab='sigma[1]', ylab='lambda[1]')
abline(h=0.025, col='gray', lty=2, lwd=2)
```



Now the posterior distribution mostly concentrates on model configurations that feature not three but rather four if not five active components. In these cases neighboring components need to collapse against each other so that their sum reconstructs one of the true components in the simulation data generating process.

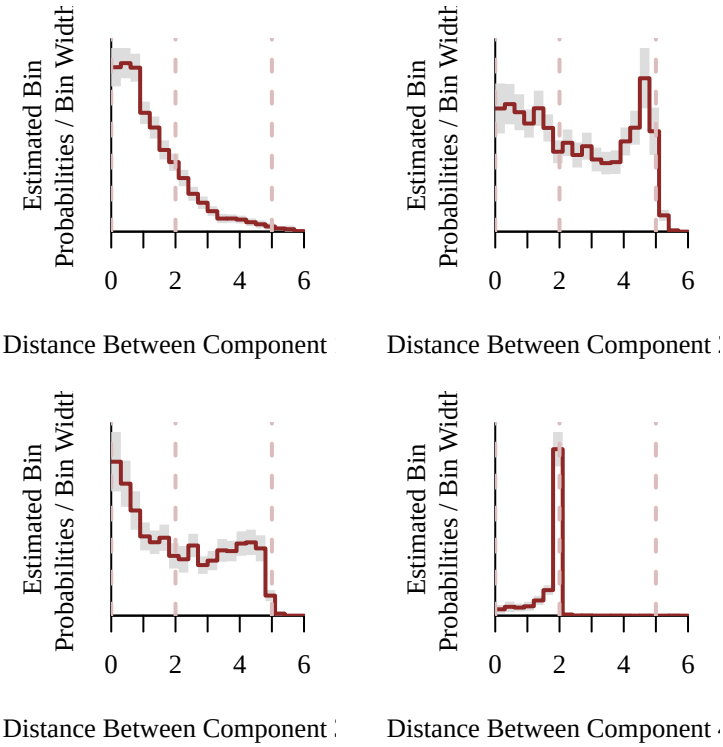
The components in simulation data generating process are centered at values of -4, 1, and 3 respectively. Consequently the distances between the true location parameters are 5 and 2. If two or more neighboring components in the mixture model collapse against each other at certain model configurations, however, then the distance between them should be 0 or as close to zero that the ordering constraint will allow.

In other words the pushforward posterior distribution for the separation between components should exhibit peaks at distances of 2 and 5. If we have extraneous components that collapse together at some model configurations then we should also see a peak at a distance of 0. Finally the location parameters of inactive components will not be coupled to the location parameters of any neighboring components, contributing a more diffuse background.

Indeed once we construct the pushforward posterior distributions for the distances we see exactly these behaviors.

```
var_repl <- list('mu'=array(sapply(1:5,
                            function(k) paste0("mu[", k, "]"))))
eval_component_separation <-
  list(function(mu) abs(mu[1] - mu[2]),
       function(mu) abs(mu[2] - mu[3]),
       function(mu) abs(mu[3] - mu[4]),
       function(mu) abs(mu[4] - mu[5]) )
names(eval_component_separation) <-
  sapply(1:4, function(k) paste0('d', k, k + 1))

component_separation <-
  util$eval_expectand_pushforwards(samples,
                                   eval_component_separation,
                                   var_repl)
```

```
par(mfrow=c(2, 2), mar=c(5, 5, 1, 1))

for (k in 1:4) {
  name <- paste0('d', k, k + 1)
  display_name <- paste('Distance Between Component', k, 'and', k + 1)
  util$plot_expectand_pushforward(component_separation[[name]],
                                  20, flim=c(0, 6),
                                  display_name=display_name)
  abline(v=0, lwd=2, lty=2, col=util$c_light)
  abline(v=2, lwd=2, lty=2, col=util$c_light)
  abline(v=5, lwd=2, lty=2, col=util$c_light)
}
```



Now that we understand the nature of these nasty degeneracies a bit better we might start to consider mediation strategies.

For example we might use a prior model for the component probabilities that concentrates on the simplex boundaries, encouraging the deactivation of any unnecessary components. That said we'd still have to contend with a degeneracy regarding which of the components are deactivated.

Similarly one approach to avoiding component collapse, indeed one that shows up over and over again in the statistics literature, is to employ *repulsive* prior models for the component locations. These prior models are designed to suppress model configurations where the component probability distributions are too close to each other but without necessarily breaking exchangeability. Consequently they have to be used in tandem with other methods, such as a sparsifying prior model for the component probabilities and/or ordering constraints.

Combining enough of these strategies together can absolutely yield reasonable results in some cases. To be honest, however, I have not found any combination of heuristics to be robust enough to make exchangeable mixture models a reliably productive tool in practice.

Beyond computational considerations exchangeable mixture models also suffer from fundamental interpretation issues. Because of the possibility of extraneous components we cannot in general associate the individual component models with explicit features of the true data

generating process. In particular we cannot interpret inferences for the component model configurations independently of the others. At that point we are not really learning about the true data generating process so much as particular patterns that happen to arise in particular observations.

For all of these reasons I avoid exchangeable mixture models as much as possible. I have found that building up mixture models from interpretable component models that can be tied to distinct aspects of the true data generating process tends to be not only more straightforward to implement but also yields more performant inferences and more generalizable predictions.

5 Conclusion

Mixture modeling is a general modeling technique that is useful in a diversity of practical applications. Here we have focused on relatively simple mixture models, but the basic structure provides a foundation for even more sophisticated models.

For example all of the examples presented in this chapter treat the component probabilities are parameters to be inferred. In some applications, however, it is more useful to *derive* them from the output of another part of the model. Amongst other features this approach allows the contribution of the component models to be mediated by external circumstances.

Similarly there's no reason why the component probabilities need to be static. In many applications it's natural for the component probabilities to vary across temporal or spatial dimensions. Modeling a sequence of evolving component probabilities results in powerful modeling techniques including the infamous hidden Markov model.

Finally individual component models can in theory be mixture models of their own. Indeed nested mixture models can be interpreted as an implementation of discrete conditional probability theory, allowing us to model data generating processes with conditional, but unobserved, logic.

Acknowledgements

A very special thanks to everyone supporting me on Patreon: Adam Fleischhacker, Adriano Yoshino, Alejandro Navarro-Martínez, Alessandro Varacca, Alex D, Alexander Noll, Alexander Rosteck, Andrea Serafino, Andrew Mascioli, Andrew Rouillard, Andrew Vigotsky, Ara Winter, Austin Rochford, Avraham Adler, Ben Matthews, Ben Swallow, Benoit Essiambre, Bertrand Wilden, Bradley Kolb, Brendan Galdo, Brynjolfur Gauti Jónsson, Cameron Smith, Canaan Breiss, Cat Shark, CG, Charles Naylor, Chase Dwelle, Chris Jones, Christopher Mehrvarzi, Colin Carroll, Colin McAuliffe, Damien Mannion, dan mackinlay, Dan W Joyce, Dan Waxman, Dan Weitzenfeld, Daniel Edward Marthaler, Daniel Saunders, Darshan Pandit, Darthmaluus , David Galley, David Wurtz, Doug Rivers, Dr. Jobo, Dr. Omri Har Shemesh, Dylan Maher, Ed

Cashin, Edgar Merkle, Eli Witus, Eric LaMotte, Ero Carrera, Eugene O'Friel, Felipe González, Fergus Chadwick, Finn Lindgren, Geoff Rollins, Håkan Johansson, Hamed Bastan-Hagh, haubur, Hector Munoz, Henri Wallen, hs, Hugo Botha, Ian, Ian Costley, idontgetoutmuch, Ignacio Vera, Ilaria Prosdocimi, Isaac Vock, Isidor Belic, J Michael Burgess, jacob pine, Jair Andrade, James C, James Hodgson, James Wade, Janek Berger, Jason Martin, Jason Pekos, jd, Jeff Burnett, Jeff Dotson, Jeff Helzner, Jeffrey Erlich, Jessica Graves, Joe Sloan, John Flournoy, Jonathan H. Morgan, Jonathon Vallejo, Joran Jongerling, JU, June, Justin Bois, Kádár András, Karim Naguib, Karim Osman, Kejia Shi, Kristian Gårdhus Wichmann, Lars Barquist, lizzie , Logan Sullivan, LOU ODETTE, Luís F, Marcel Lüthi, Marek Kwiatkowski, Mariana Carmona, Mark Donoghoe, Markus P., Márton Vaitkus, Matthew, Matthew Kay, Matthew Mulvahill, Matthieu LEROY, Mattia Arsendi, Maurits van der Meer, Michael Colaresi, Michael DeWitt, Michael Dillon, Michael Lerner, Mick Cooney, Mike Lawrence, N Sanders, N.S. , Name, Nathaniel Burbank, Nic Fishman, Nicholas Clark, Nicholas Cowie, Nick S, Octavio Medina, Ole Rogeberg, Oliver Crook, Olivier Ma, Patrick Kelley, Patrick Boehnke, Pau Pereira Batlle, Peter Johnson, Pieter van den Berg, ptr, Ramiro Barrantes Reynolds, Raúl Peralta Lozada, Ravin Kumar, Rémi , Rex Ha, Riccardo Fusaroli, Richard Nerland, Robert Frost, Robert Goldman, Robert kohn, Robin Taylor, Ryan Grossman, Ryan Kelly, S Hong, Sean Wilson, Sergiy Protsiv, Seth Axen, shira, Simon Duane, Simon Lilburn, Spencer Carter, sssz, Stan_user, Stephen Lienhard, Stew Watts, Stone Chen, Susan Holmes, Svilup, Tao Ye, Tate Tunstall, Tatsuo Okubo, Teresa Ortiz, Theodore Dasher, Thomas Siegert, Thomas Vladeck, Tobychev, Tomáš Frýda, Tony Wuersch, Virginia Fisher, Vladimir Markov, Wil Yegelwel, Will Farr, Will Lowe, Will^2, woejozney, Xianda Sun, yolhaj , yureq , Zach A, Zad Rafi, and Zhengchen Ca.

License

A repository containing all of the files used to generate this chapter is available on GitHub.

Original Computing Environment

```
writeLines(readLines(file.path(Sys.getenv("HOME"), ".R/Makevars")))
```

```
CC=clang

CXXFLAGS=-O3 -mtune=native -march=native -Wno-unused-variable -Wno-unused-function -Wno-macr
CXX=clang++ -arch x86_64 -ftemplate-depth-256

CXX14FLAGS=-O3 -mtune=native -march=native -Wno-unused-variable -Wno-unused-function -Wno-ma
CXX14=clang++ -arch x86_64 -ftemplate-depth-256
```

```
sessionInfo()
```

```
R version 4.3.2 (2023-10-31)
Platform: x86_64-apple-darwin20 (64-bit)
Running under: macOS Sonoma 14.4.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRlapack.dylib;

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] colormap_0.1.4     rstan_2.32.6       StanHeaders_2.32.7

loaded via a namespace (and not attached):
 [1] gtable_0.3.4       jsonlite_1.8.8     compiler_4.3.2     Rcpp_1.0.11
 [5] stringr_1.5.1      parallel_4.3.2     gridExtra_2.3      scales_1.3.0
 [9] yaml_2.3.8         fastmap_1.1.1      ggplot2_3.4.4      R6_2.5.1
[13] curl_5.2.0         knitr_1.45         tibble_3.2.1       munsell_0.5.0
[17] pillar_1.9.0       rlang_1.1.2        utf8_1.2.4         V8_4.4.1
[21] stringi_1.8.3      inline_0.3.19      xfun_0.41          RcppParallel_5.1.7
[25] cli_3.6.2          magrittr_2.0.3     digest_0.6.33      grid_4.3.2
```

```
[29] lifecycle_1.0.4    vctrs_0.6.5        evaluate_0.23      glue_1.6.2
[33] QuickJSR_1.0.8     codetools_0.2-19   stats4_4.3.2       pkgbuild_1.4.3
[37] fansi_1.0.6        colorspace_2.1-0   rmarkdown_2.25     matrixStats_1.2.0
[41] tools_4.3.2        loo_2.6.0          pkgconfig_2.0.3    htmltools_0.5.7
```

Stan
Program 2 `signal\_background1.stan`

```
data {
  // Signal and background observations
  int<lower=1> N;
  array[N] real<lower=0> y;
}

parameters {
  real mu_signal;                // Signal location
  real<lower=0> sigma_signal;    // Signal scale
  real<lower=0> beta_back;       // Background scale
  real<lower=0, upper=1> lambda; // Background probability
}

model {
  // Prior model
  mu_signal ~ normal(50, 50 / 2.32);   // 0 <~ mu_signal    <~ 100
  sigma_signal ~ normal(0, 25 / 2.57); // 0 <~ sigma_signal <~  25
  beta_back ~ normal(0, 50 / 2.57);    // 0 <~ beta_back    <~  50
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    target += log_mix(lambda,
                      exponential_lpdf(y[n] | 1 / beta_back),
                      cauchy_lpdf(y[n] | mu_signal, sigma_signal));
  }
}

generated quantities {
  array[N] real<lower=0> y_pred = rep_array(-1, N);

  for (n in 1:N) {
    if (bernoulli_rng(lambda)) {
      y_pred[n] = exponential_rng(1 / beta_back);
    } else {
      while (y_pred[n] < 0) {
        y_pred[n] = cauchy_rng(mu_signal, sigma_signal);
      }
    }
  }
}
```

Stan
Program 3 `signal\_background2.stan`

```
data {
  // Signal and background observations
  int<lower=1> N;
  array[N] real<lower=0> y;
}

parameters {
  real mu_signal;                // Signal location
  real<lower=0> sigma_signal;    // Signal scale
  real<lower=0> beta_back;       // Background scale
  real<lower=0, upper=1> lambda; // Background probability
}

model {
  // Prior model
  mu_signal ~ normal(50, 50 / 2.32);   // 0 <~ mu_signal    <~ 100
  sigma_signal ~ normal(0, 25 / 2.57); // 0 <~ sigma_signal <~  25
  beta_back ~ normal(0, 50 / 2.57);    // 0 <~ beta_back    <~  50
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    target += log_mix(lambda,
                      exponential_lpdf(y[n] | 1 / beta_back),
                      cauchy_lpdf(y[n] | mu_signal, sigma_signal));
  }
}

generated quantities {
  array[N] real<lower=0, upper=1> p;
  array[N] int<lower=0, upper=1> z_pred;

  for (n in 1:N) {
    vector[2] xs = [   log(lambda)
                     + exponential_lpdf(y[n] | 1 / beta_back),
                       log(1 - lambda)
                     + cauchy_lpdf(y[n] | mu_signal, sigma_signal) ]';
    p[n] = softmax(xs)[1];
    z_pred[n] = bernoulli_rng(p[n]);
  }
}
```

Stan
Program 4 `simu\_zip.stan`

```
data {
  int<lower=1> N;   // Number of observations
  real<lower=0> mu; // Poisson intensity
  real<lower=0, upper=1> lambda; // Main component probability
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] int<lower=0> y = rep_array(0, N);

  for (n in 1:N) {
    // If we sample the non-inflating component then replace initial
    // value with a Poisson sample
    if (bernoulli_rng(lambda)) {
      y[n] = poisson_rng(mu);
    }
  }
}
```

Stan
Program 5 `zip1.stan`

```
data {
  int<lower=1> N;          // Number of observations
  array[N] int<lower=0> y; // Positive integer observations
}

parameters {
  real<lower=0> mu;              // Poisson intensity
  real<lower=0, upper=1> lambda; // Main component probability
}

model {
  // Prior model
  mu ~ normal(0, 15 / 2.57); // 0 <~ mu <~ 15
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    if (y[n] == 0) {
      target += log_mix(lambda, poisson_lpmf(y[n] | mu), 0);
    } else {
      target += log(lambda) + poisson_lpmf(y[n] | mu);
    }
  }
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] int<lower=0> y_pred = rep_array(0, N);

  for (n in 1:N) {
    // If we sample the non-inflating component then replace initial
    // value with a Poisson sample
    if (bernoulli_rng(lambda)) {
      y_pred[n] = poisson_rng(mu);
    }
  }
}
```

Stan
Program 6 `prior\_tune.stan`

```
functions {
  // Differences between inverse gamma tail
  // probabilities and target probabilities
  vector tail_delta(vector y, vector theta,
                    array[] real x_r, array[] int x_i) {
    vector[2] deltas;
    deltas[1] = inv_gamma_cdf(theta[1] | exp(y[1]), exp(y[2])) - 0.01;
    deltas[2] = 1 - inv_gamma_cdf(theta[2] | exp(y[1]), exp(y[2])) - 0.01;
    return deltas;
  }
}

data {
  real<lower=0>     y_low;
  real<lower=y_low> y_high;
}

transformed data {
  // Initial guess at inverse gamma parameters
  vector[2] y_guess = [log(2), log(5)]';
  // Target quantile
  vector[2] theta = [y_low, y_high]';
  vector[2] y;
  array[0] real x_r;
  array[0] int x_i;

  // Find inverse Gamma density parameters that ensure
  // 1% probability below y_low and 1% probability above y_high
  y = algebra_solver(tail_delta, y_guess, theta, x_r, x_i);

  print("alpha = ", exp(y[1]));
  print("beta = ", exp(y[2]));
}

generated quantities {
  real alpha = exp(y[1]);
  real beta = exp(y[2]);
}
```

Stan
Program 7 `zip2.stan`

```
data {
  int<lower=1> N;          // Number of observations
  array[N] int<lower=0> y; // Positive integer observations
}

parameters {
  real<lower=0> mu;              // Poisson intensity
  real<lower=0, upper=1> lambda; // Main component probability
}

model {
  // Prior model
  mu ~ inv_gamma(3.5, 9.0); // 1   <~   mu   <~ 15
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    if (y[n] == 0) {
      target += log_mix(lambda, poisson_lpmf(y[n] | mu), 0);
    } else {
      target += log(lambda) + poisson_lpmf(y[n] | mu);
    }
  }
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] int<lower=0> y_pred = rep_array(0, N);

  for (n in 1:N) {
    // If we sample the non-inflating component then replace initial
    // value with a Poisson sample
    if (bernoulli_rng(lambda)) {
      y_pred[n] = poisson_rng(mu);
    }
  }
}
```

Stan
Program 8 `simu\_zoib.stan`

```
data {
  int<lower=1> N; // Number of observations
}

transformed data {
  real alpha = 3;
  real beta = 2;
  simplex[3] lambda = [0.75, 0.15, 0.10]';
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] real<lower=0, upper=1> y;

  for (n in 1:N) {
    int z = categorical_rng(lambda);

    if (z == 1) {
      y[n] = beta_rng(alpha, beta);
    } else if (z == 2) {
      y[n] = 0;
    } else {
      y[n] = 1;
    }
  }
}
```

Stan
Program 9 `zoib1.stan`

```
data {
  int<lower=1> N;                     // Number of observations
  array[N] real<lower=0, upper=1> y; // Unit-interval valued observations
}

transformed data {
  int<lower=0> N_zero = 0;
  int<lower=0> N_one = 0;
  int<lower=0> N_else = N;

  for (n in 1:N) {
    if (y[n] == 0) N_zero += 1;
    if (y[n] == 1) N_one  += 1;
  }

  N_else -= N_one + N_zero;
}

parameters {
  real<lower=0> alpha; // Beta shape
  real<lower=0>  beta; // Beta scale
  simplex[3]   lambda; // Component probabilities
}

model {
  // Prior model
  alpha ~ normal(0, 10 / 2.57); // 0 <~ alpha <~ 10
  beta ~ normal(0, 10 / 2.57);  // 0 <~ beta  <~ 10
  // Implicit uniform prior density function for lambda

  // Observational model
  target += multinomial_lpmf({N_else, N_zero, N_one} | lambda);

  for (n in 1:N) {
    if (0 < y[n] && y[n] < 1) {
      target += beta_lpdf(y[n] | alpha, beta);
    }
  }
}

generated quantities {
  array[N] real<lower=0, upper=1> y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);

    if (z == 1) {
      y_pred[n] = beta_rng(alpha, beta);
    } else if (z == 2) {
      y_pred[n] = 0;
```

Stan
Program 10 `zoib2a.stan`

```
data {
  // Number of non-zero/one observations
  int<lower=1> N_else;
  // Non-zero/one observations
  array[N_else] real<lower=0, upper=1> y_else;
}

parameters {
  real<lower=0> alpha; // Beta shape
  real<lower=0>  beta; // Beta scale
}

model {
  // Prior model
  alpha ~ normal(0, 10 / 2.57); // 0 <~ alpha <~ 10
  beta ~ normal(0, 10 / 2.57);  // 0 <~ beta  <~ 10

  // Observational model
  target += beta_lpdf(y_else | alpha, beta);
}
```

Stan
Program 11 `zoib2b.stan`

```
data {
  int<lower=1> N_zero; // Number of zero observations
  int<lower=1> N_one;  // Number of one observations
  int<lower=1> N_else; // Number of non-zero/one observations
}

parameters {
  simplex[3] lambda; // Component probabilities
}

model {
  // Prior model
  // Implicit uniform prior density function for lambda

  // Observational model
  target += multinomial_lpmf({N_else, N_zero, N_one} | lambda);
}
```

Stan
Program 12 simu_normal_mix.stan

```
data {
  int<lower=1> N;      // Number of observations
}

transformed data {
  int K = 3;                                      // Number of components
  array[K] real mu = {-4, 1, 3};                  // Component locations
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
  simplex[K] lambda = [0.3, 0.5, 0.2]';           // Component probabilities

}

generated quantities {
  array[N] real y;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 13 `normal\_mix1.stan`

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                        // Number of components
  array[K] real mu = {-4, 1, 3};                    // Component locations
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 14 normal_mix2a.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                        // Number of components
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  array[K] real mu;  // Component locations
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32); // -10 <~ mu[k] <~ +10
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 15 normal_mix2b.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                      // Number of components
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  array[K] real mu;  // Component locations
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  mu[1] ~ normal(-4, 2 / 2.32); // -6 <~ mu[2] <~ -2
  mu[2] ~ normal( 0, 2 / 2.32); // -2 <~ mu[2] <~ +2
  mu[3] ~ normal(+4, 2 / 2.32); // +2 <~ mu[2] <~ +6
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 16 normal_mix2c.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                        // Number of components
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  ordered[K] mu;     // Component locations
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32); // -10 <~ mu[k] <~ +10
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 17 `normal\_mix3a.stan`

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3; // Number of components
}

parameters {
  array[K] real mu;              // Component locations
  array[K] real<lower=0> sigma; // Component scales
  simplex[K] lambda;             // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32);    // -10 <~  mu[k]   <~ +10
  sigma ~ normal(0, 10 / 2.57); //   0 <~ sigma[k] <~ +10

  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 18 normal_mix3b.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3; // Number of components
}

parameters {
  ordered[K] mu;                // Component locations
  array[K] real<lower=0> sigma; // Component scales
  simplex[K] lambda;            // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32);    // -10 <~  mu[k]   <~ +10
  sigma ~ normal(0, 10 / 2.57); //   0 <~ sigma[k] <~ +10

  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 19 `normal\_mix4.stan`

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations

  int K; // Number of components
}

parameters {
  ordered[K] mu;                // Component locations
  array[K] real<lower=0> sigma; // Component scales
  simplex[K] lambda;            // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32);    // -10 <~  mu[k]   <~ +10
  sigma ~ normal(0, 10 / 2.57); //   0 <~ sigma[k] <~ +10

  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```