```
html"""<style>
main {
    margin: 0 auto;
    max-width: 90%;
    padding-left: max(50px, 1%);
    padding-right: max(253px, 10%);
    # 253px to accomodate TableOfContents(aside=true)
}
"""
```

```
begin
    using Markdown
    using InteractiveUtils
    using Pkg, DrWatson, PlutoUI
end
```

```
TaskLocalRNG()
```

```
begin
    using Distributions
    using FillArrays
    using StatsPlots

    using LinearAlgebra
    using Random
    using Turing

    # Import all libraries.
    using DataFrames
    #DirichletProcess, ChineseRestaurantProcess,
    StickBreakingProcess
    using Turing.RandomMeasures
    # DocStringExtensions provides $(SIGNATURES)
    using DocStringExtensions
    # Set a random seed.
    Random.seed!(3)
end
```

```
using DynamicPPL, Printf
```

```
using Plots
# Plot the cluster assignments over time
```

Table of Contents

```
begin
  PlutoUI.TableOfContents()
end
```

```julia
versioninfo()
```

```
Julia Version 1.10.2
Commit bd47eca2c8a (2024-03-01 10:14 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 32 × Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-15.0.7 (ORCJIT, haswell)
Threads: 16 default, 0 interactive, 8 GC (on 32 virtua
l cores)
Environment:
  JULIA_PKG_SERVER = https://mirrors.tuna.tsinghua.ed
u.cn/julia
  JULIA_REVISE_WORKER_ONLY = 1
```

Mixture models

- https://turinglang.org/dev/tutorials/06-infinite-mixture-model/

1 Two-component mixture model.

- Check Gaussian `mixture.jl` or .ipynb for more.

```
two_comp_mixture (generic function with 2 methods)
```

```
@model function two_comp_mixture(x)
    # Hyper-parameters
    μ0 = 0.0
    σ0 = 1.0

    # Draw weights.
    π1 ~ Beta(1, 1)
    π2 = 1 - π1

    # Draw locations of the components.
    μ1 ~ Normal(μ0, σ0)
    μ2 ~ Normal(μ0, σ0)

    # Draw latent assignment.
    z ~ Categorical([π1, π2])

    # Draw observation from selected component.
    if z == 1
        x ~ Normal(μ1, σ0)
    else
        x ~ Normal(μ2, σ0)
    end
end
```

2 Finite Mixture Model

If we have more than two components, this model can elegantly be extended using a Dirichlet distribution as prior for the mixing weights, $\pi_1, \ldots, \pi_K$. Note that the Dirichlet distribution is the multivariate generalization of the beta distribution. The resulting model can be written as:

$$(\pi_1, \ldots, \pi_K) \sim Dirichlet(K, \alpha)$$

$$\mu_k \sim Normal(\mu_0, \Sigma_0), \forall k$$

$$z \sim Categorical(\pi_1, \ldots, \pi_K)$$

$$x \sim Normal(\mu_z, \Sigma)$$

which resembles the model in the Gaussian mixture model tutorial with a slightly different notation.

```
md"# 2  Finite Mixture Model

If we have more than two components, this model can elegantly be extended using a Dirichlet distribution as prior for the mixing weights, $\pi_1, ..., \pi_K$. Note that the Dirichlet distribution is the multivariate
generalization of the beta distribution. The resulting
model can be written as:

$(\pi_1, ..., \pi_K) \sim Dirichlet(K, \alpha)$

$\mu_k \sim Normal(\mu_0, \Sigma_0), \forall k$

$z \sim Categorical(\pi_1, ..., \pi_K)$

$x \sim Normal(\mu_z, \Sigma)$

which resembles the model in the [Gaussian mixture model tutorial](https://turinglang.org/stable/tutorials/01-gaussian-mixture-model/) with a slightly different notation."
```

3 Infinite Mixture Model

The question now arises, is there a generalization of a Dirichlet distribution for which the dimensionality is infinite, i.e. $K=\infty$?

But first, to implement an infinite Gaussian mixture model in Turing, we first need to load the `Turing.RandomMeasures` module. RandomMeasures contains a variety of tools useful in nonparametrics.

```
md"# 3 Infinite Mixture Model

The question now arises, is there a generalization of a Dirichlet distribution for which the dimensionality is
infinite, i.e. $K=\infty$?

But first, to implement an infinite Gaussian mixture model in Turing, we first need to load the `Turing.RandomMeasures` module. RandomMeasures contains a variety of tools useful in nonparametrics."
```

We utilize the fact that one can integrate out the mixing weights in a Gaussian mixture model allowing us to arrive at the Chinese restaurant process construction. See Carl E. Rasmussen: The Infinite Gaussian Mixture Model, NIPS (2000) for details.

In fact, if the mixing weights are integrated out, the conditional prior for the latent variable is given by:

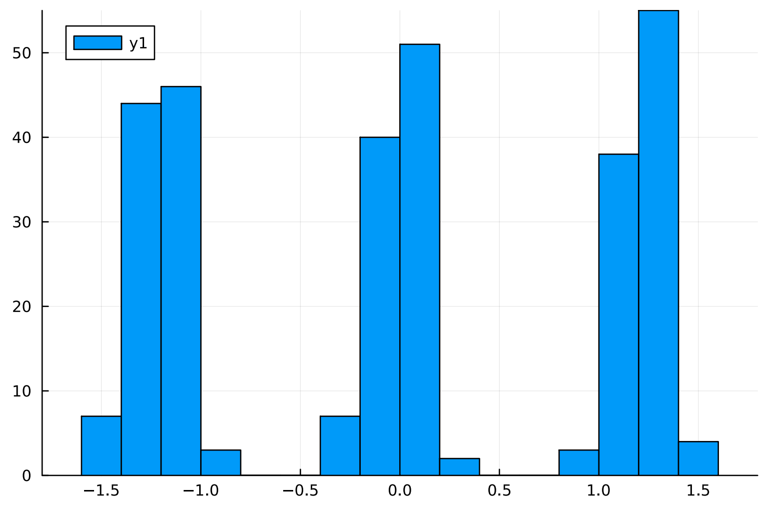
$$p(z_i=k|z_{\neg i}, \alpha) = \frac{n_k + \alpha K}{N - 1 + \alpha}$$

where z_{-i} are the latent assignments of all observations except observation i. Note that we use n_k to denote the number of observations at component k excluding observation i. The parameter α is the concentration parameter of the Dirichlet distribution used as prior over the mixing weights.

```
md"We utilize the fact that one can integrate out the mixing weights in a Gaussian mixture model allowing us to arrive at the Chinese restaurant process construction. See
Carl E. Rasmussen: The Infinite Gaussian Mixture Model,
NIPS (2000) for details.

In fact, if the mixing weights are integrated out, the
conditional prior for the latent variable is given by:

$p(z_i=k|z_{\neg i}, \alpha) = \frac{n_k + \alpha K}{ N - 1
+ \alpha}$


where $z_{-i}$ are the latent assignments of all observations except observation $i$. Note that we use $n_k$ to denote the number of observations at component $k$ excluding observation $i$. The parameter $\alpha$ is the concentration parameter of the Dirichlet distribution used as prior over the mixing weights."
```

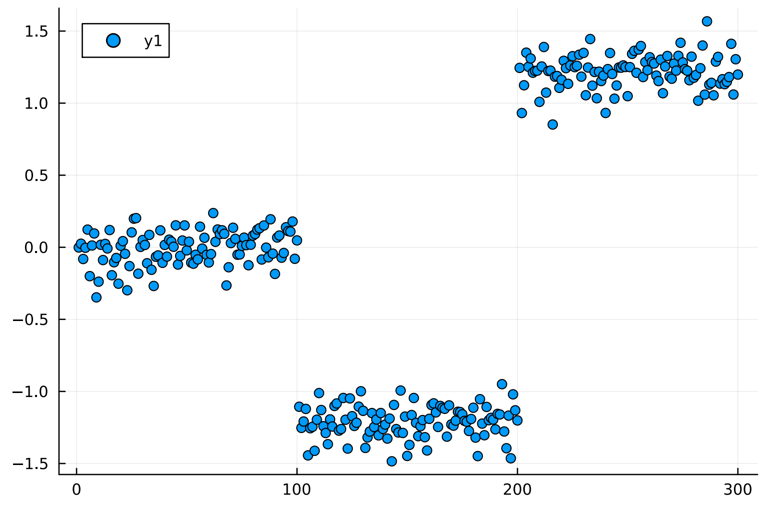
3.0 Simulate a 3-cluster Gaussian mixture data

We simulate/create some random data that comes from three clusters, with means of 0, -10, and 10, which is a bit easier than to infer 0, -5, 10.



```
begin
    # Generate some test data.
    Random.seed!(1)
    # cluster size
    csize=100
    data_gmm = vcat(randn(csize), randn(csize) .- 10,
    randn(csize) .+ 10)
    @show size(data_gmm)
    @show mean(data_gmm)
    # normalize data
    data_gmm .-= mean(data_gmm)
    data_gmm /= std(data_gmm);
    # draw histogram
    histogram(data_gmm, bins=20)
end
```

```
size(data_gmm) = (300,)
mean(data_gmm) = 0.07469478561903005
```



```
# plot 3 clusters of simulated data
scatter(1:csize*3, data_gmm)
```

```
view(::Vector{Int64}, 201:300): [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
```

```julia
begin
    z_truth_vec = ones(Int, csize*3);
    z_truth_vec[101:200] .= 2
    z_truth_vec[201:300] .= 3
end
```

3.1 Chinese Restaurant Process

To obtain the Chinese restaurant process construction, we can now derive the conditional prior if $K \to \infty$.

For $n_k > 0$, we obtain:

$$p(z_i = k | z_{\neg i}, \alpha) = \frac{n_k}{N - 1 + \alpha}$$

and for all infinitely many clusters that are empty (combined) we get:

$$p(z_i = k | z_{\neg i}, \alpha) = \frac{\alpha}{N - 1 + \alpha}$$

Those equations show that the conditional prior for component assignments is proportional to the number of such observations, meaning that the Chinese restaurant process has a rich get richer property.

To get a better understanding of this property, we can plot the cluster choosen by for each new observation drawn from the conditional prior.

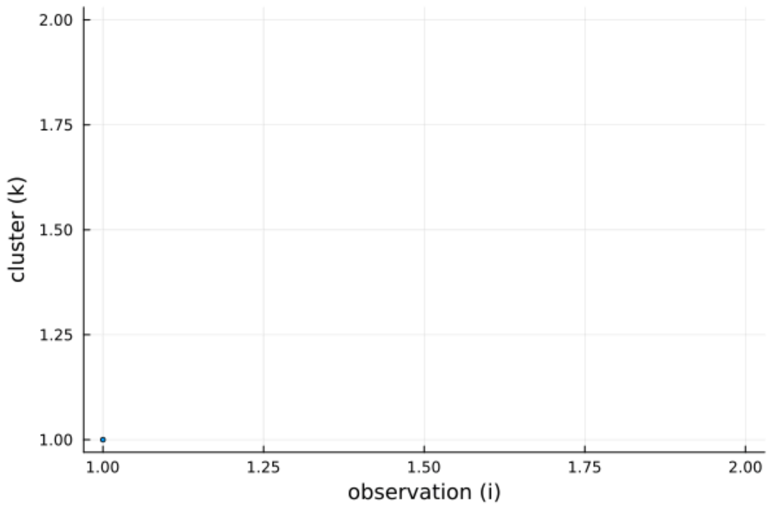
```
Turing.RandomMeasures
```

```julia
begin
    @show parentmodule(DirichletProcess)
    @show parentmodule(ChineseRestaurantProcess)
    @show parentmodule(StickBreakingProcess)
end
```

```
parentmodule(DirichletProcess) = Turing.RandomMeasu
res
parentmodule(ChineseRestaurantProcess) = Turing.Random
Measures
parentmodule(StickBreakingProcess) = Turing.RandomMeas
ures
```

3.1.1 Visualize the cluster growth given the growing data points

```julia
begin
    # Concentration parameter.
    α = 1.0

    # Random measure, e.g. Dirichlet process.
    rpm = DirichletProcess(α)

    # Cluster assignments for each observation.
    z = Vector{Int}()

    # Maximum number of observations we observe.
    Nmax = 500

    for i in 1:Nmax
        # Number of observations per cluster.
        K = isempty(z) ? 0 : maximum(z)
        nk = Vector{Int}(map(k -> sum(z .== k), 1:K))

        # Draw new assignment.
        push!(z, rand(ChineseRestaurantProcess(rpm, nk)))
    end
end
```



```julia
@gif for i in 1:Nmax
    scatter(
        collect(1:i),
        z[1:i];
        markersize=2,
        xlabel="observation (i)",
        ylabel="cluster (k)",
        legend=false,
    )
end
```

```
Saved animation to /tmp/jl_rK7SnUxGny.gif
```

```
[121, 220, 144, 8, 4, 3]
```

```julia
begin
    # number of samples in each cluster
    no_of_clusters = maximum(z)
    nk=Vector{Int}(map(k -> sum(z .==k), 1:no_of_clusters ))
end
```

```
1
```

```julia
rand(ChineseRestaurantProcess(rpm, nk))
```

Further, we can see that the number of clusters is logarithmic in the number of observations and data points. This is a side-effect of the `rich-get-richer` phenomenon, i.e. we expect large clusters and thus the number of clusters has to be smaller than the number of observations.

$$E[K|N] \approx \alpha * log(1 + \frac{N}{\alpha})$$

We can see from the equation that the concentration parameter α allows us to control the number of clusters formed *a priori*.

In Turing we can implement an infinite Gaussian mixture model using the Chinese restaurant process construction of a Dirichlet process as follows:

3.1.2 CRP on a 3-cluster-Gaussian-Mixture-generated data

```
md"### 3.1.2 CRP on a 3-cluster-Gaussian-Mixture-generated
data"
```

```
infiniteGMM (generic function with 4 methods)
```

```julia
@model function infiniteGMM(x, warn=true)
    # Hyper-parameters, i.e. concentration parameter and parameters of H.
    α = 0.5  # the smaller α is, the less clusters a priori.
    μ0 = 0.0  # μ of Prior to draw Gaussian cluster mean
    σ0 = 1.0  # σ of Prior to draw Gaussian cluster mean
    σ1 = 1.0  # σ for each Gaussian cluster

    # Define random measure, e.g. Dirichlet process.
    rpm = DirichletProcess(α)

    # The base distribution, to draw the mean value of all Gaussian distributions in the Dirichlet process.
    H = Normal(μ0, σ0)

    # Latent assignment.
    z = zeros(Int, length(x)) # tzeros() = zeros()

    # Locations of the infinitely many Gaussian clusters.
    μ = zeros(Float64, 0)
    # Number of clusters.
    K = 0

    for i in 1:length(x)

        # Number of clusters.
        #K = maximum(z)
        nk = Vector{Int}(map(k -> sum(z .== k), 1:K))

        # Draw the latent assignment.
        z[i] ~ ChineseRestaurantProcess(rpm, nk)

        # Create a new cluster?
        if z[i] > K
            K += 1
            push!(μ, 0.0)

            # Draw location of new cluster.
            μ[z[i]] ~ H
        end

        # An observation that follows this distribution
        x[i] ~ Normal(μ[z[i]], σ1)
    end
end
```

```
[0, 0, 0]
```

```julia
# test: tzeros() seems to be identical to zeros(). not sure why the original example uses tzeros()
tzeros(Int, 3)
```

We can now use Turing to infer the assignments of some data points.

	iteration	chain	z[1]	z[2]	z[3]	z[4]	
1	1	1	1.0	1.0	1.0	1.0	1
2	2	1	1.0	1.0	1.0	1.0	1
3	3	1	1.0	1.0	1.0	1.0	1
4	4	1	1.0	1.0	1.0	1.0	1
5	5	1	1.0	1.0	1.0	1.0	1
6	6	1	1.0	1.0	1.0	1.0	1
7	7	1	1.0	1.0	1.0	1.0	1
8	8	1	1.0	1.0	1.0	1.0	1
9	9	1	1.0	1.0	1.0	1.0	1
10	10	1	1.0	1.0	1.0	1.0	1
more							

```julia
begin
    # Fit InfiniteGMM (CRP-based) model to the simulated data above
    Random.seed!(2)
    # compile the model
    @time model_infini_GMM_CRP = infiniteGMM(data_gmm);
    # sample 1500 iterations via the Sequential Monte Carlo sampler.
    @time chain_infini_GMM_CRP = sample(model_infini_GMM_CRP, SMC(), 1500);
end
```

```
100%
```

```
  0.000000 seconds
 55.146279 seconds (86.20 M allocations: 11.144 GiB, 5.93% gc time)
```

	iteration	chain	z[1]	z[2]	z[3]	z[4]	
1	1496	1	1.0	1.0	1.0	1.0	1.
2	1497	1	1.0	1.0	1.0	1.0	1.
3	1498	1	1.0	1.0	1.0	1.0	1.
4	1499	1	1.0	1.0	1.0	1.0	1.
5	1500	1	1.0	1.0	1.0	1.0	1.

```julia
begin
    # turn the MCMC chains/estimates into a dataframe
    @time chain_infini_GMM_CRP_DF = DataFrame(chain_infini_GMM_CRP)
    #@show first(chain_infini_GMM_CRP_DF,5)
    last(chain_infini_GMM_CRP_DF, 5)
end
```

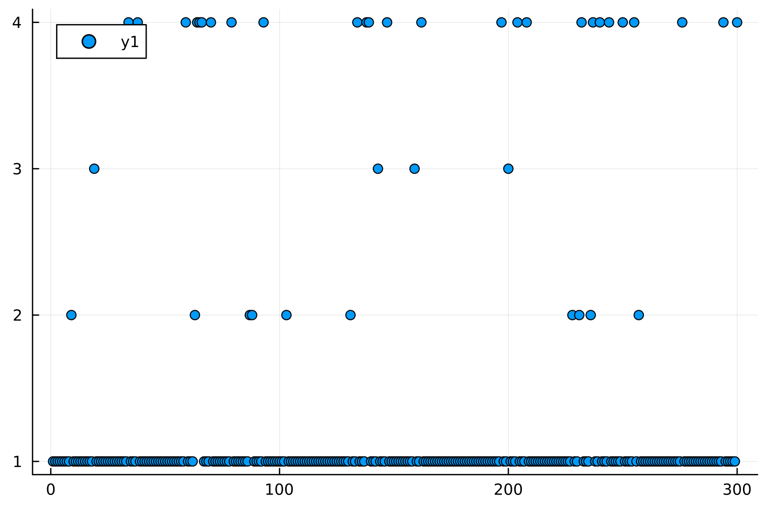
```
  0.007099 seconds (4.65 k allocations: 8.340 MiB)
```

```
["iteration", "chain", "z[1]", "z[2]", "z[3]", "z[4]", "z[5]", ":
```

```julia
names(chain_infini_GMM_CRP_DF)
```

```
(1500, 310)
```

```julia
size(chain_infini_GMM_CRP_DF)
```



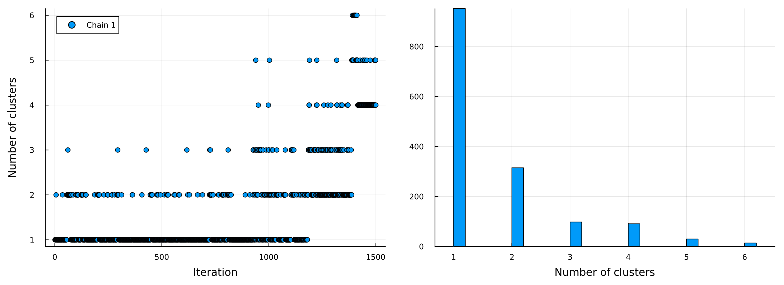
```julia
begin
    #plot the estimated z (cluster membership) vs data
    index (do not look good)
    z_estimated = vec(chain_infini_GMM_CRP[1500,
    MCMCChains.namesingroup(chain_infini_GMM_CRP,
    :z),:].value)
    scatter(1:csize*3, z_estimated)
end
```

```
[1.0, 2.0, 3.0, 4.0]
```

```julia
# return the unique clusters discovered by infini_GMM_CRP
unique(z_estimated)
```

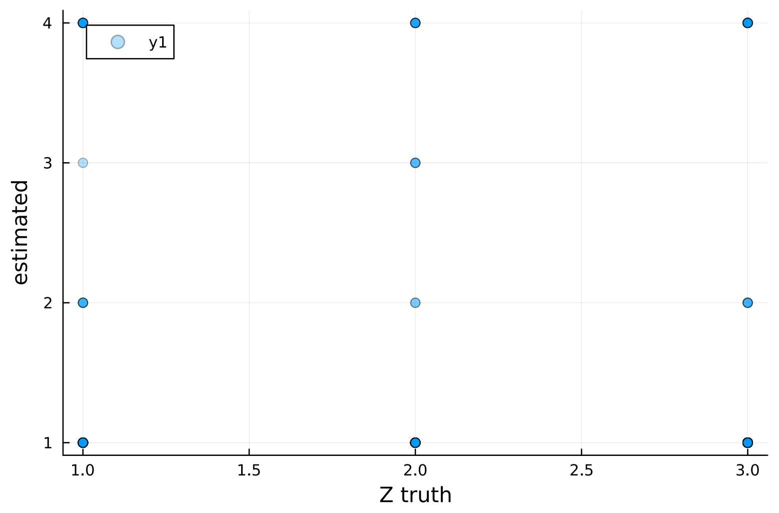
```
draw_cluster_no_in_chain (generic function with 1 method)
```

```julia
function draw_cluster_no_in_chain(chain; burnin_iter=1,
iterations=1000)
    # Extract the number of clusters for each sample of the
Markov chain.
    K = map(
        t -> length(unique(vec(chain[t,
MCMCChains.namesingroup(chain, :z), :].value))),
        burnin_iter:iterations,
    );

    # Visualize the number of clusters.
    s_of_K = scatter(burnin_iter:iterations, K;
xlabel="Iteration",
        ylabel="Number of clusters", label="Chain 1")
    h_of_K = histogram(K; xlabel="Number of clusters",
legend=false)

    plot(s_of_K, h_of_K, layout=(1,2),
left_margin=5*Plots.mm,
        bottom_margin=5*Plots.mm, size=(1200,430) )
end
```



```
@time draw_cluster_no_in_chain(chain_infini_GMM_CRP;
burnin_iter=1, iterations=1500)
```

```
0.322596 seconds (2.85 M allocations: 190.556 Mi
B, 14.13% gc time)
```



```
scatter(z_truth_vec , z_estimated, alpha=0.3, xlab="Z
truth", ylab="estimated")
```

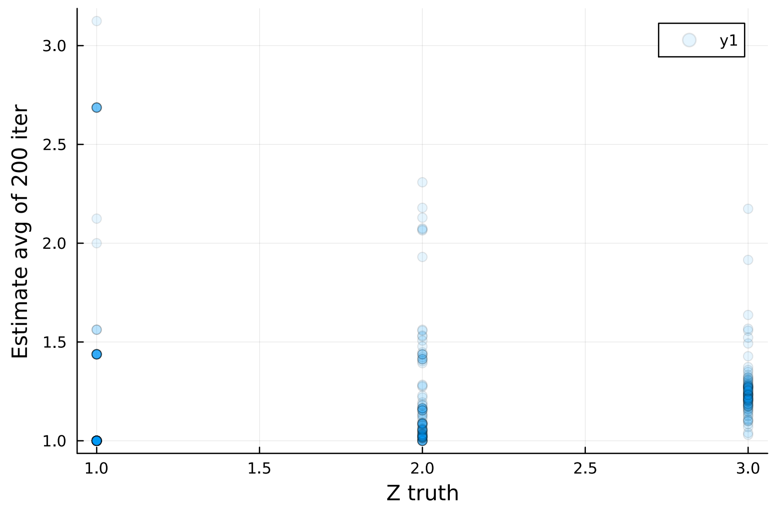
```
201×300 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  2.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     2.0  2.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  2.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  2.0  1
 ⋮                        ⋮              ⋱                 ⋮
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  2.0  3.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  4
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  4.0  4.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  4.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  4.0  1.0  1.0  1
```

```
begin

    z_estimate_chain_value =
    Array(chain_infini_GMM_CRP[1300:1500,
    MCMCChains.namesingroup(chain_infini_GMM_CRP, :z),:])
    z_estimate_ar = reshape(z_estimate_chain_value, (201,
    300))
end
```

```
@time z_estimate_avg =
  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.56219, 1.0, 1.0, 1.0,
```

```
@time z_estimate_avg = mean.(eachcol(z_estimate_ar))
```

```
0.000102 seconds (3 allocations: 2.562 KiB)
```



```
scatter(z_truth_vec , z_estimate_avg, alpha=0.1, xlab="Z truth", ylab="Estimate avg of 200 iter")
```

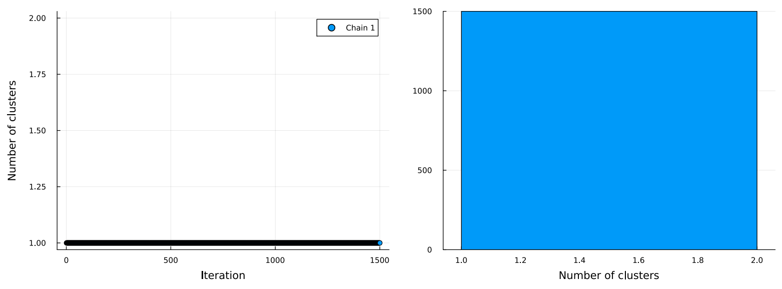
3.1.3 Use a PG+HMC Gibbs sampler

```
md"### 3.1.3 Use a PG+HMC Gibbs sampler"
```

```
# k, μ, w are all mapped into the sampler?
@time chain_infini_GMM_CRP_via_Gibbs = sample(model_infini_GMM_CRP,  Gibbs(PG(100, :z), HMC(0.05, 10, :μ)), 1500);
```

```
100%
```

```
4348.623424 seconds (6.72 G allocations: 722.361 GiB, 3.47% gc time, 0.19% compilation time)
```



```
@time draw_cluster_no_in_chain(chain_infini_GMM_CRP_via_Gibbs; burnin_iter=1, iterations=1500)
```

```
  0.549389 seconds (2.87 M allocations: 191.821 MiB, 34.84% gc time, 16.57% compilation time)
```

3.2 Stick-breaking process

$$\beta_k \sim Beta(1, \alpha)$$

$$\pi_k = \beta_k \Pi_{l=1}^{k-1}(1 - \beta_l)$$

$$\theta_k^* \sim H$$

$$G = \Sigma_{k=1}^{\infty} \pi_k \delta_{\theta_k^*}$$

- no further testing.

3.3 Size biased sampling

`infiniteGMM_size_biased`

Arguments:

- x: 1D data.
- rpm: is a Random process measure. e.g. Dirichlet process.

```
rpm = DirichletProcess(0.5)
```

- Hyper-parameters, i.e. concentration parameter and parameters of H.

~~~julia

"""
$(SIGNATURES)
Arguments:
- x: 1D data.
- rpm: is a Random process measure. e.g. Dirichlet process.

```julia
rpm = DirichletProcess(0.5)
~~~

- Hyper-parameters, i.e. concentration parameter and
parameters of H.

"""
@model function infiniteGMM_size_biased(x, rpm)
    # The base distribution, to draw the mean value of all Gaussian distributions in the Dirichlet process.
    H = Normal(0.0, 2)

    N = length(x)
    # Latent assignment.
    z = zeros(Int, N) #was tzeros()

    # Locations of the infinitely many clusters.
    μ = zeros(Float64, 0)

    # probability weights for each class
    J = zeros(Float64, N)

    K = 0
    surplus=1.0
    for i in 1:N
        ps = vcat(J[1:K], surplus)
        z[i] ~ Categorical(ps)

        # Create a new cluster?
        if z[i] > K
            K += 1
            push!(μ, 0.0)
            # Assign a weight
            J[K] ~ SizeBiasedSamplingProcess(rpm, surplus)
            μ[K] ~ H
            surplus -= J[K]
        end

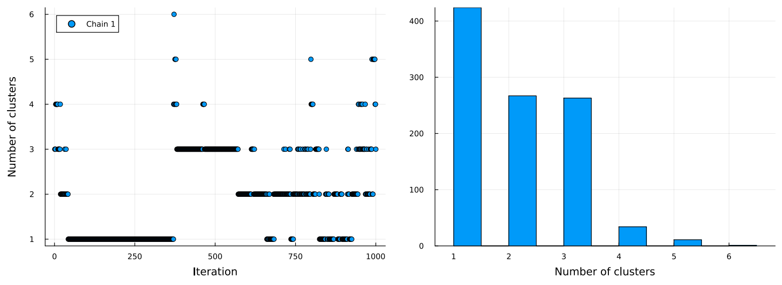
        # An observation that follows this distribution
        x[i] ~ Normal(μ[z[i]], 1)
    end
end
```
```

	iteration	chain	z[1]	z[2]	z[3]	z[4]	
1	1	1	1.0	1.0	1.0	1.0	1
2	2	1	1.0	1.0	1.0	1.0	1
3	3	1	1.0	1.0	1.0	1.0	1
4	4	1	1.0	1.0	1.0	1.0	1
5	5	1	1.0	1.0	1.0	1.0	1
6	6	1	1.0	1.0	1.0	1.0	1
7	7	1	1.0	1.0	1.0	1.0	1
8	8	1	1.0	1.0	1.0	1.0	1
9	9	1	1.0	1.0	1.0	1.0	1
10	10	1	1.0	1.0	1.0	1.0	1
	more						

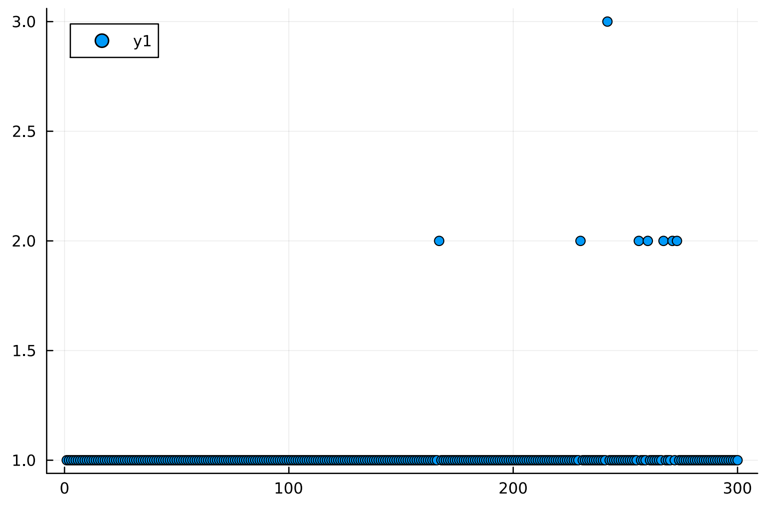
```
begin
    # Fit the simulated GMM data with an size_biased
    infinite GMM
    Random.seed!(2)
    @time model_size_biased =
    infiniteGMM_size_biased(data_gmm,
    DirichletProcess(0.5));
    # MCMC via Sequential Monte Carlo
    @time chain_infini_GMM_size_biased =
    sample(model_size_biased, SMC(), 1000);
end
```

```
100%
```

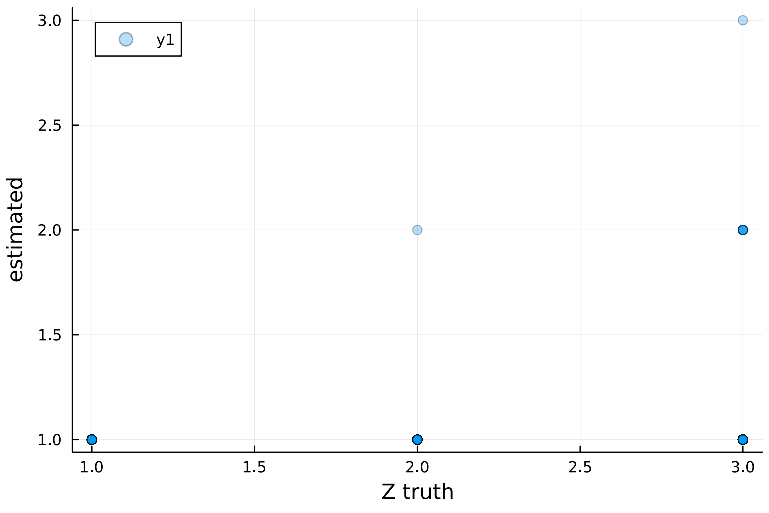
```
  0.000000 seconds
 31.720661 seconds (52.98 M allocations: 5.094 GiB, 5.
28% gc time)
```



```
draw_cluster_no_in_chain(chain_infini_GMM_size_biased)
```



```julia
begin
    z_estimated_2 = vec(chain_infini_GMM_size_biased[1000,
    MCMCChains.namesingroup(chain_infini_GMM_size_biased,
    :z),:].value)
    scatter(1:300, z_estimated_2)
end
```



```julia
scatter(z_truth_vec , z_estimated_2, alpha=0.3, xlab="Z
truth", ylab="estimated")
```

```
Enter cell code...
```

Conclusions

- The infinite mixture models did not fit well.

```
md"# Conclusions

- The infinite mixture models did not fit well."
```