ZIPBN: Zero-inflated Poisson Bayesian Networks for Zero-inflated Count Data

Junsouk Choi

Dec 7, 2019

Introduction

In this vignette, we will give a tutorial on using `ZIPBN`. `ZIPBN` fits zero-inflated Poisson Bayesian networks (ZIPBN) for zero-inflated count data, such as scRNA-seq data, using Markov chain Monte Carlo (MCMC). The ZIPBN models deal with conditional dependencies of variables of zero-inflated count data. The model and its MCMC implementation are intermediate results of on-going work on scalable Bayesian networks for the scRNA-seq data. Therefore, any manuscripts are not available yet, which explain the ZIPBN model and how to implement it. Before showing how to use `ZIPBN` to make a network analysis on zero-inflated count data, we briefly introduce the ZIPBN model and its MCMC schema.

Zero-inflated Poisson Bayesian Networks

The goal is to discover conditional dependence structures from zero-inflated count data using Bayesian networks. A Bayesian network, also known as directed acyclic graph (DAG), is a pair $\{V, A\}$, where $V = \{1, \ldots, p\}$ is a set of nodes representing random variables $\mathbf{X} = \{X_1, \ldots, X_p\}$ and $A = (a_{jk})$ is a set of directed edges with $a_{jk} = 1$ meaning a directed edge $k \to j$ from the node k to the node j. We often call A as the adjacency matrix. Beyesian networks allow no cycle, meaning one cannot return to the same node by following the directed edges. The acyclicity of Bayesian networks leads to a factorization of the joint distribution of $\mathbf{X}$ into a set of local distributions, $p(\mathbf{X}|A) = \prod_{j=1}^{p} p(X_j|X_{pa(j)})$. Here, $pa(j) = \{k \in V | a_{jk} = 1\}$ is a set of parents of node j and $X_{pa(j)} = \{X_k : k \in pa(j)\}$ is a subset of random variables indexed by $pa(j)$.

To deal with excessive zeros in zero-inflated count data, we model each local distribution in the factorization to be a zero-inflated Poisson model as following:

$$Pr(X_j = x|\mathbf{X}_{pa(j)}) = \begin{cases} \pi_j + (1-\pi_j)Poi(0|\mu_j) & \text{if} \quad x = 0 \\ (1-\pi_j)Poi(x|\mu_j) & \text{if} \quad x > 0. \end{cases}$$

where $\pi_j = \text{logit}^{-1}(\sum_{k=1}^{p} \alpha_{jk} X_k + \delta_j)$ and $\mu_j = \exp(\sum_{k=1}^{p} \beta_{jk} X_k + \gamma_j)$ with $\alpha_{jk} = \beta_{jk} = 0$ if $k \notin pa(j)$. We denote the proposed ZIPBN model by $ZIPBN(\alpha, \beta, \delta, \gamma)$, where parameters are $\alpha = (\alpha_{jk})$, $\beta = (\beta_{jk})$, $\delta = (\delta_j)$, and $\gamma = (\gamma_j)$ with $\alpha_{jk} = \beta_{jk} = 0$ if $k \notin pa(j)$.

We adopt a Bayesian inference strategy to make inference on the ZIPBN models. Inference on parameters α and β which allows sparsity naturally leads to the selection of the effective edges. A spike-and-slab prior is imposed on each element of α and β as following:

$$\alpha_{jk}|a_{jk}, \tau_\alpha \sim a_{jk} N\left(0, \tau_\alpha^{-1}\right) + (1 - a_{jk}) N\left(0, (\nu\tau_\alpha)^{-1}\right),$$
$$\beta_{jk}|a_{jk}, \tau_\beta \sim a_{jk} N\left(0, \tau_\beta^{-1}\right) + (1 - a_{jk}) N\left(0, (\nu\tau_\beta)^{-1}\right),$$

where ν is sufficiently large. Notice that a_{jk} represents whether an edge $k \to j$ is selected based on the data. Furthermore, we assume that δ_j and γ_j follow Normal prior distributions with mean 0 and precisions τ_δ and τ_γ respectively. For the graph parameter a_{jk}, we use a Bernoulli prior with success probability ρ. The hierarchical formulation of our model is completed by assigning a Gamma prior to each of τ's and a Beta prior to ρ.

For posterior inference, we sample parameters from the posterior distributions using an MCMC algorithm. The MCMC algorithm updates each parameter by Gibbs sampler at each iteration. When the full conditional distribution is not available in closed form, we update it through a Metropolis step. The most difficult part of the MCMC implementation is to sample a_{jk}'s, due to their ugly posterior probability space. To overcome it, we sample a_{jk} jointly with α_{jk}, β_{jk}, δ_j, and γ_j. Additionally, one of two proposal strategies is chosen with probability of 0.5. The first one is that if there exist (or doesn't exist) an edge $k \to j$, a Metropolis sampler proposes addition of the edge (or deletion of the edge). The second strategy is to propose reversing the edge for a Metropolis step. This MCMC algorithm is implemented as `ZIPBN` package.

Generating Example Data

We generate example data using ZIPBN models, rather than loading real data. It will be helpful to understand what the ZIPB model is, since procedure of data generation gives a clear sketch of ZIPBN models. First, we load `ZIPBN` package and randomly generate a graph. `igraph` package is also loaded to make use of its convenient functions for analyzing graph structures.

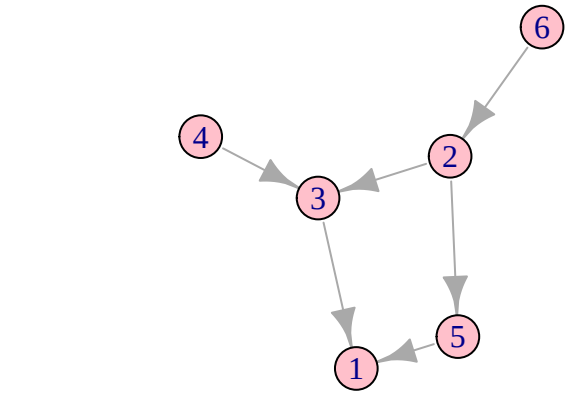
```
# load ZIPBN package with igraph package
library(ZIPBN)
library(igraph)
```

```
# set random seed for reproducibility
set.seed(20191207)

# randomly generate a DAG with p nodes and p - 1 edges
p = 6
n_edges = p
A = matrix(0, p, p)
while (sum(A == 1) < n_edges)
{
   id_edge = matrix(sample(1 : p, 2), ncol = 2)
   A[id_edge] = 1
   g = graph_from_adjacency_matrix(t(A))

   # if selected edge make a directed cycle, discard the edge
   if (!is_dag(g))
      A[id_edge] = 0
}

# show the generated DAG
A
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    0    0    1    0    1    0
#> [2,]    0    0    0    0    0    1
#> [3,]    0    1    0    1    0    0
#> [4,]    0    0    0    0    0    0
#> [5,]    0    1    0    0    0    0
#> [6,]    0    0    0    0    0    0
plot.igraph(g, vertex.size = 25, vertex.color = "pink",
            main = "DAG generated with 6 nodes and 6 edges")
```



A DAG with 6 nodes and 6 edges is generated. `graph_from_adjacency_matrix` function creates an `igraph` graph object from an adjacency matrix and `is_dag` function tests whether the given `igraph` graph is a DAG. The above code randomly suggests a directed edge and tests whether the suggested edge makes a directed cycle. If a directed cycle is made, we discard the proposed edge. It is repeated until the graph has the prespecified number of edges. Notice that ZIPBN models define an adjacency matrix as $A = (a_{jk})$ with $a_{jk} = 1$ if $k \to j$, but in the case of `igraph` , $a_{jk} = 1$ if $j \to k$. We, therefore, need to transpose our adjacency matrix before converting it to an `igraph` graph. The adjacency matrix `A` we generate is printed and its graph is plotted using `plot.igraph` function of `igraph`. We then generate data from a ZIPBN model depending on the DAG we create.

```
# set parameters of the ZIPBN model, given DAG A
alpha = matrix(0, p, p)
alpha[A == 1] = 0.2
beta  = matrix(0, p, p)
beta[A == 1]  = -0.2
delta = rep(-1, p)
gamma = rep(2, p)

# generate data from the ZIPBN model with true parameters
n = 1000
x = matrix(0, n, p)
order_nodes = as_ids(topo_sort(g))
order_nodes
#> [1] 4 6 2 3 5 1
for (j in order_nodes)
{
  # calculate pi_j
  pi = exp(x %*% alpha[j, ] + delta[j])
  pi = pi / (1 + pi)
  # calculate mu_j
```

```
  mu = exp(x %*% beta[j, ] + gamma[j])
  # generate data for X_j
  x[ , j] = rpois(n, mu) * (1 - rbinom(n, 1, pi))
}


head(x)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    0    1    4    0    4    5
#> [2,]    0    0    5    0    3   11
#> [3,]    0    6    3    0    1    0
#> [4,]    0    0    1   10    7    4
#> [5,]    0    2    0    6    8    5
#> [6,]    0    0    0    9    3    0
mean(x == 0)   # proprtion of zeros
#> [1] 0.4453333
```

Given the generated adjacency matrix (DAG) `A`, parameters are specified for ZIPBN. If there is a directed edge $k \to j$, the corresponding α_{jk} and β_{jk} equal to 0.2 and -0.2 respectively. Otherwise, they are zeros. All δ_j's has the value of -1, while γ_j's are 2. We draw 1000 samples from the ZIPBN model with the specified parameters. Here, some functions of `igraph` are used. `topo_sort` function orders the nodes of a DAG graph so that each node comes before all nodes to which it has edges. `as_ids` function convert a node sequence to an ordinary vector. 4, 6, 2, 3, 5, 1 are the order of our DAG and we generate data in the sequence of this order. Observe that the proportion of zeros in data is 0.4453333, which indicates the generated data are sparse with excessive zeros.

Using `ZIPBN` for Example Data

In this section, we demonstrate how `ZIPBN` can be used to discover conditional dependence structures of variables from zero-inflated count data. `ZIPBN` consists of a function `mcmc_ZIPBN` which implements the MCMC for ZIPBN models. `mcmc_ZIPBN` needs three essential arguments `starting`, `tuning`, and `priors`.

```
# create starting value list
m = colMeans(x)
v = apply(x, 2, var)
starting = list(alpha = matrix(0, p, p),
                beta  = matrix(0, p, p),
                delta = log((v - m) / (m * m)),
                gamma = log((v - m + m * m) / m),
                A     = matrix(0, p, p),
                tau   = c(10, 10, 1, 1),
                rho   = 0.1)
```

`starting` is a list with each tag corresponding to a parameter name. The value portion of each tag is the parameters' starting values for MCMC. Valid tags are alpha, beta, delta, gamma, A, tau, and rho. If you have any prior information about graph structure of your data, it would be better to incorporate the information into the starting values. We assume we know nothing about our data. The empty graph is a good choice for the starting value of A, since we believe a graph is sparse. The starting values of alpha and beta should be zero matrices accordingly. For the starting values of delta and gamma, assuming all variables are independent, we calculate the plug-in estimates for δ and γ. Values between 1 and 10 are good enogh for Markov chains of τ's to start. The choice of starting values of tau does not affect much to the performance of our MCMC. The value of 0.1 is reasonable for the starting value of rho, due to sparsity assumption for graph structure.

```
# create tuning value list
tuning = list(phi_alpha = c(1e+8, 20),
              phi_beta  = c(1e+8, 100),
              phi_delta = 5,
              phi_gamma = 50,
              phi_A     = c(1e+10, 10, 10, 1, 10))
```

`tuning` is a list with each tag corresponding to a parameter name. The value portion of each tag defines the precision of Normal proposal distribution for Metropolis sampler. Valid tags are phi_alpha, phi_beta, phi_delta, phi_gamma, and phi_A. phi_alpha and phi_beta should be a positive vector of length 2. When a new value is proposed for each element, if there is no edge corresponding to the element, the Metropolis sampler uses the first element as the precision of the proposal distribution. Otherwise, the second element is used. phi_A must be a positive vector of length 5. Note that an element of A is proposed jointly with the corresponding element of alpha, beta, delta, and gamma. The third and fourth elements are precisions of Normal proposal distributions for δ and γ. The first element is used for proposal regarding α and β, if deletion of an edge is proposed. The second and third elements are used to propose a new value of each element of α and β, when proposing addition of an edge. The tuning values of phi_alpha, phi_beta, phi_delta, and phi_gamma should be chosen by monitoring the acceptance rates. The desired acceptance rate is between 25% and 60%. Furthermore, we need to set values of phi_A for the MCMC sampler to report neither too few nor too many changes of edges.

```
# create priors list
priors = list(nu        = 10000^2,
              tau_alpha = c(0.01, 0.01),
              tau_beta  = c(0.01, 0.01),
              tau_delta = c(0.01, 0.01),
              tau_gamma = c(0.01, 0.01),
              rho       = c(0.5, 0.5))
```

`priors` is a list with each tag corresponding to a parameter name. The value portion of each tag defines hyperparameters of the priors specified for ZIPBN models Valid tags are nu, tau_alpha, tau_beta, tau_delta, tau_gamma, and rho. The value of nu controls the spike part of our spike-and-slab prior for α and β's. Our experiment shows that 10000^2 or 1000^2 gives satisfactory results. tau_alpha, tau_beta, tau_delta, are tau_gamma define hyperparameters of Gamma priors for precisions of Normal priors on ZIPBN parameters. Values of 0.01, 0.01 are most commonly used, since the hyperparameter values makes Gamma prior uninformative. rho determines hyperparameters of Beta prior on the inclusion probability of edges. We set , as it assign high probability near 0 and we assume sparsity for graph structure.

```
# run mcmc_ZIPBN function
n_sample = 5000
n_burnin = 2500
verbose  = TRUE
n_report = 1000
out = mcmc_ZIPBN(x, starting, tuning, priors, n_sample, n_burnin, verbose, n_report)
#> An edge 6 -> 2 is added
#> An edge 1 -> 3 is added
#> An edge 6 -> 1 is added
#> An edge 4 -> 3 is added
#> An edge 2 -> 3 is added
#> An edge 5 -> 1 is added
#> An edge 6 -> 1 is deleted
#> An edge 2 -> 5 is added
#> An edge 4 -> 1 is added
#>
#> iter= 1000
```

```
#> acceptance rates of alpha:
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]  0.0 25.5 26.7  7.3  9.2 26.1
#> [2,] 25.9  0.0 24.3 24.0 25.2  7.5
#> [3,] 16.3 18.6  0.0  8.4 25.2 25.6
#> [4,] 25.3 22.8 25.8  0.0 25.2 26.6
#> [5,] 26.0 13.3 26.6 24.7  0.0 24.8
#> [6,] 24.6 26.7 23.6 25.7 22.6  0.0
#> acceptance rates of beta:
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]  0.0 31.7 26.4  5.4 13.2 25.5
#> [2,] 23.4  0.0 26.5 25.5 26.3  9.8
#> [3,] 23.3 20.9  0.0 13.0 25.6 25.6
#> [4,] 24.8 25.0 28.5  0.0 24.7 25.6
#> [5,] 24.6 14.1 25.4 23.9  0.0 26.6
#> [6,] 24.8 25.8 24.0 25.3 24.7  0.0
#> acceptance rates of delta:
#> [1] 17.8 23.1 23.4 19.3 20.4 19.1
#> acceptance rates of gamma:
#> [1] 17.8 23.1 23.4 19.3 20.4 19.1
#>
#> An edge 1 -> 3 is reversed to 3 -> 1
#> An edge 4 -> 1 is deleted
#>
#> iter= 2000
#> acceptance rates of alpha:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 27.60 25.00 12.35 10.30 26.80
#> [2,] 28.15  0.00 26.80 27.25 27.40  7.70
#> [3,] 21.85 17.90  0.00  8.20 26.45 28.50
#> [4,] 28.40 25.95 27.45  0.00 27.90 26.70
#> [5,] 27.60 12.95 28.80 27.15  0.00 27.15
#> [6,] 27.05 27.10 27.30 27.95 26.15  0.00
#> acceptance rates of beta:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 28.35 25.25 11.65 12.55 25.10
#> [2,] 23.35  0.00 26.60 25.50 25.65  9.30
#> [3,] 24.15 21.40  0.00 13.00 26.10 25.25
#> [4,] 25.55 24.60 26.45  0.00 25.00 25.25
#> [5,] 24.50 13.85 24.65 24.20  0.00 25.65
#> [6,] 24.85 25.00 24.75 25.95 24.90  0.00
#> acceptance rates of delta:
#> [1] 19.10 23.20 23.65 19.15 20.90 19.30
#> acceptance rates of gamma:
#> [1] 19.10 23.20 23.65 19.15 20.90 19.30
#>
#> An edge 6 -> 1 is added
#> An edge 6 -> 1 is deleted
#>
#> iter= 3000
#> acceptance rates of alpha:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 30.00 22.70 19.80  9.97 28.23
```

```
#> [2,] 30.37  0.00 29.00 29.43 29.07  7.47
#> [3,] 24.50 17.87  0.00  8.47 28.97 30.23
#> [4,] 29.50 27.20 29.30  0.00 28.93 28.53
#> [5,] 29.83 13.00 29.57 28.90  0.00 30.00
#> [6,] 29.27 29.47 28.97 30.03 28.93  0.00
#> acceptance rates of beta:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 26.80 25.77 15.77 12.57 24.10
#> [2,] 24.03  0.00 26.33 25.57 25.27  9.77
#> [3,] 24.00 20.60  0.00 12.93 25.70 25.53
#> [4,] 25.93 24.57 26.93  0.00 25.27 25.20
#> [5,] 24.27 13.80 24.43 24.90  0.00 24.97
#> [6,] 24.87 25.13 24.73 25.63 25.83  0.00
#> acceptance rates of delta:
#> [1] 19.60 21.83 23.50 19.07 20.43 19.60
#> acceptance rates of gamma:
#> [1] 19.60 21.83 23.50 19.07 20.43 19.60
#>
#>
#> iter= 4000
#> acceptance rates of alpha:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 29.88 21.73 22.07  9.60 28.75
#> [2,] 30.88  0.00 29.05 29.78 29.50  7.27
#> [3,] 25.75 18.07  0.00  8.38 29.40 30.68
#> [4,] 29.25 27.50 29.55  0.00 29.75 29.28
#> [5,] 30.23 12.90 29.33 29.55  0.00 30.18
#> [6,] 30.23 29.55 29.58 30.33 29.65  0.00
#> acceptance rates of beta:
#>       [,1]  [,2]  [,3] [,4]  [,5]  [,6]
#> [1,]  0.00 27.77 26.55 18.6 12.30 25.00
#> [2,] 24.75  0.00 26.40 25.8 25.80  9.68
#> [3,] 25.62 20.50  0.00 13.0 26.38 26.35
#> [4,] 26.85 24.77 27.57  0.0 26.52 25.97
#> [5,] 25.52 13.45 24.85 26.0  0.00 26.12
#> [6,] 26.15 25.82 25.50 26.1 26.50  0.00
#> acceptance rates of delta:
#> [1] 20.45 21.27 23.45 19.68 20.30 19.32
#> acceptance rates of gamma:
#> [1] 20.45 21.27 23.45 19.68 20.30 19.32
#>
#>
#> iter= 5000
#> acceptance rates of alpha:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 29.98 21.30 22.98  9.78 29.14
#> [2,] 31.14  0.00 29.40 29.66 29.92  7.32
#> [3,] 27.00 17.94  0.00  8.64 29.80 30.88
#> [4,] 29.56 27.52 29.18  0.00 29.62 29.48
#> [5,] 30.36 12.88 29.64 29.76  0.00 29.88
#> [6,] 30.28 30.00 28.92 30.06 29.24  0.00
#> acceptance rates of beta:
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
```

```
#> [1,]  0.00 27.62 26.20 20.70 12.58 25.00
#> [2,] 25.46  0.00 26.56 26.24 25.54  9.88
#> [3,] 26.62 20.60  0.00 12.76 26.58 26.74
#> [4,] 26.56 25.70 27.16  0.00 26.96 25.84
#> [5,] 25.98 13.16 25.70 25.92  0.00 26.16
#> [6,] 26.16 25.92 25.52 25.64 26.52  0.00
#> acceptance rates of delta:
#> [1] 20.60 21.64 23.66 19.92 20.50 19.46
#> acceptance rates of gamma:
#> [1] 20.60 21.64 23.66 19.92 20.50 19.46
```

We run the `mcmc_ZIPBN` function with the generated data in the previous section. `starting`, `tuning`, and `priors` are passed into the `mcmc_ZIPBN` function. `n_sample = 5000` and `n_burin = 2500` are specified to determine the number of MCMC iterations and burn-in samples.
Since we set `verbose = TRUE` and `n_report = 1000`, progress of the MCMC sampler and Metropolis acceptance rates are reported every 1000th interation. Observe that the printed acceptance rates are between 25% and 60%, which indicates our MCMC sampler is tuned well.

Posterior Inference via `ZIPBN` Results

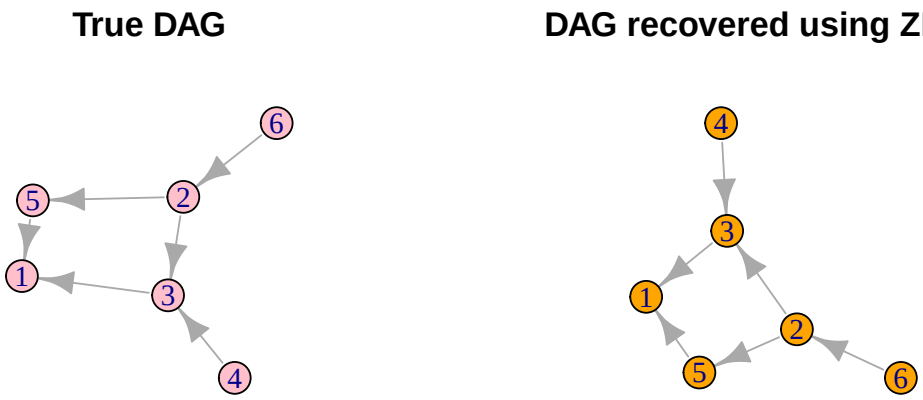
The `mcmc_ZIPBN` function returns MCMC samples from posterior distributions for the defined parameters of ZIPBN models as well as Metropolis acceptance percents. We are going to do posterior inference using the MCMC samples in this section.

```
# report Metropolis sampling acceptance percents
out$acceptance
#> $alpha
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 29.98 21.30 22.98  9.78 29.14
#> [2,] 31.14  0.00 29.40 29.66 29.92  7.32
#> [3,] 27.00 17.94  0.00  8.64 29.80 30.88
#> [4,] 29.56 27.52 29.18  0.00 29.62 29.48
#> [5,] 30.36 12.88 29.64 29.76  0.00 29.88
#> [6,] 30.28 30.00 28.92 30.06 29.24  0.00
#>
#> $beta
#>       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]  0.00 27.62 26.20 20.70 12.58 25.00
#> [2,] 25.46  0.00 26.56 26.24 25.54  9.88
#> [3,] 26.62 20.60  0.00 12.76 26.58 26.74
#> [4,] 26.56 25.70 27.16  0.00 26.96 25.84
#> [5,] 25.98 13.16 25.70 25.92  0.00 26.16
#> [6,] 26.16 25.92 25.52 25.64 26.52  0.00
#>
#> $gamma
#> [1] 20.60 21.64 23.66 19.92 20.50 19.46
#>
#> $delta
#> [1] 20.60 21.64 23.66 19.92 20.50 19.46
```

The Metropolis sampling acceptance rates seems good, as mentioned in the previous section. We then make inference on graph structure.

```
# recover garph structure
cutoff = 0.5
A_est  = 1 * (apply(out$samples$A, c(1, 2), mean) > cutoff)

# show the generated DAG
A_est
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    0    0    1    0    1    0
#> [2,]    0    0    0    0    0    1
#> [3,]    0    1    0    1    0    0
#> [4,]    0    0    0    0    0    0
#> [5,]    0    1    0    0    0    0
#> [6,]    0    0    0    0    0    0
g_est = graph_from_adjacency_matrix(t(A_est))
par(mfrow = c(1, 2))
plot.igraph(g, vertex.size = 25, vertex.color = "pink", main = "True DAG")
plot.igraph(g_est, vertex.size = 25, vertex.color = "orange", main = "DAG recovered using ZIPBN")
```



The posterior mean of each element a_{jk} of A represents probability that the edge $k \to j$ exists. To recover the graph, therefore, we need to set a cutoff to determine which edges are selected. Here, graph structure is recovered by applying the cutoff 0.5. We print the recovered adjacency matrix A_est. Its graph is also plotted with the graph of true DAG by using `plot.igraph` function. Notice that the underlying graph structure of true DAG is completely found by our estimate.

```
# collect MCMC samples corresponding to the recovered graph
subset = apply(out$samples$A == array(A_est, dim = c(p, p, n_sample - n_burnin)), 3, all)
alpha_mcmc = out$samples$alpha[ , , subset]
beta_mcmc  = out$samples$beta[ , , subset]
delta_mcmc = out$samples$delta[ , subset]
gamma_mcmc = out$samples$gamma[ , subset]

# calculate the posterior mean of each parameter, given the recovered graph
alpha_est = apply(alpha_mcmc, c(1, 2), mean)
beta_est  = apply(beta_mcmc, c(1, 2), mean)
```

```
delta_est = rowMeans(delta_mcmc)
gamma_est = rowMeans(gamma_mcmc)

# report the posterior mean of each parameter
round(alpha_est, digits = 2)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    0 0.00 0.26 0.00 0.21 0.00
#> [2,]    0 0.00 0.00 0.00 0.00 0.22
#> [3,]    0 0.23 0.00 0.21 0.00 0.00
#> [4,]    0 0.00 0.00 0.00 0.00 0.00
#> [5,]    0 0.22 0.00 0.00 0.00 0.00
#> [6,]    0 0.00 0.00 0.00 0.00 0.00
round(beta_est, digits = 2)
#>      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,]    0  0.00 -0.18  0.00 -0.18  0.00
#> [2,]    0  0.00  0.00  0.00  0.00 -0.21
#> [3,]    0 -0.19  0.00 -0.21  0.00  0.00
#> [4,]    0  0.00  0.00  0.00  0.00  0.00
#> [5,]    0 -0.21  0.00  0.00  0.00  0.00
#> [6,]    0  0.00  0.00  0.00  0.00  0.00
round(delta_est, digits = 2)
#> [1] -0.93 -1.28 -1.28 -0.93 -1.12 -1.01
round(gamma_est, digits = 2)
#> [1] 1.96 1.96 1.98 2.03 2.00 2.01
```

Provided the estimated DAG, we continue posterior inference on other ZIPBN parameters. First thing we need to do is to collect MCMC samples which correspond to the recovered graph. Based on the subset of MCMC samples, the posterior mean of each parameter is calculated. Observe that our estimates fairly match with true values of parameters.

```
# calculate the highest posterior density intervals for ZIPBN parameters
library(HDInterval)
hdi(alpha_mcmc[1, 3, ], credMass = 0.95)   # credible interval for \alpha_{13}
#>     lower     upper
#> 0.1995487 0.3571684
#> attr(,"credMass")
#> [1] 0.95
hdi(beta_mcmc[1, 3, ], credMass = 0.95)    # credible interval for \beta_{13}
#>      lower      upper
#> -0.2321712 -0.1420238
#> attr(,"credMass")
#> [1] 0.95
hdi(delta_mcmc[1, ], credMass = 0.95)      # credible interval for \delta_{1}
#>      lower      upper
#> -1.1558605 -0.7445094
#> attr(,"credMass")
#> [1] 0.95
hdi(gamma_mcmc[1, ], credMass = 0.95)      # credible interval for \gamma_{1}
#>    lower    upper
#> 1.904136 2.003261
#> attr(,"credMass")
#> [1] 0.95
```

In addition to point estimates, we are able to construct a credible interval for each parameter based on the posterior samples. The credible interval is a Bayesian alternative to the confidence interval. The package

`HDInterval` calculates the highest posterior density intervals using samples from the posterior distribution. The highest posterior density interval is the credible interval with the shortest length. Here, we calculate 95% highest posterior density intervals for α_{13}, β_{13}, δ_1, and γ_1. All of the intervals contain the true parameter values.