package mcleod: Monte Carlo for Estimating Latent Over-Dispersion

Barak Brill, Daniel Yekutieli

June 6, 2022

Contents

1 Introduction

The `mcleod` package allows the user to model the mixing distribution of binomial and Poisson mixtures. Specifically, let $X_i, i = 1, ..., n$, denote n binomial samples distributed via $X_i \sim bin(N_i, P_i)$, where the N_i's are some known constants and the P_i's are i.i.d. random variables from an unknown distribution. The `mcleod` package allows users to estimate the distribution of P_i's and construct confidence intervals for its quantiles and CDF values. For the Poisson case, the mixing distribution is modeled using $X_i \sim Pois(\lambda_i)$, where the λ_i's are drawn from an unknown distribution estimated from the data.

In addition, the package allows users to fit a random intercept model of the following form: $X_i \sim bin(N_i, P_i)$; $log(P_i/(1-P_i)) = \gamma_i + \vec{\beta}^T \vec{Z}_i$, where γ_i is an intercept term drawn from general distribution, whose CDF is estimated from the data, $\vec{Z}_i$ is a vector of sample covariates, and $\vec{\beta}$ is a vector of slope coefficients, also estimated from the data. For the Poisson case, the mcleod package allows for the following model: $X_i \sim Pois(\lambda_i)$; $log(\lambda_i) = \gamma_i + \vec{\beta}^T \vec{Z}_i$.

The document is structured as follows. In Section 2 we introducing the generative model for the setting with no covariates, and briefly explain how the mixing distribution is estimated. In Section 3 we show how point estimates for the CDF of the mixing distribution can be obtained using the package. In Section 4 we briefly discuss how confidence intervals for the quantiles and percentiles of the mixing distribution are constructed and show several code examples. In Section 5 we discuss the different random intercept models implemented in the `mcleod` package and provide example for fitting those models to data. In Section 6 we discuss some advanced topics.

2 The generative model

We present the approach for modeling the mixing distribution in the binomial case, where $X_i \sim bin(N_i, P_i)$ and and the density of $log(P_i/(1-P_i))$ is estimated from the data. The approach for modeling Poisson samples is similar, and involves modeling the distribution of $log(\lambda_i)$ instead.

Let $\gamma_i \equiv log(P_i/(1-P_i))$. The P_i's are not observed (only indirectly through the X_i's). We take an hierarchical Bayesian approach for modeling distribution of P_i's. Let I be some complete power of 2, e.g., 32 or 64. Let $\vec{a} = (a_0, a_1, ..., a_I), a_0 < a_1 < ... < a_I$, be a vector with $I+1$ entries marking $I+1$ partitioning point on the real line. We approximate the density function for the distribution of γ_i's using a piece-wise constant density function, with "jumps" at the $I-1$ internal points of $\vec{a}$. This density function is characterized by I unique function values, which we will denote using the vector $\vec{\pi} = (\pi_1, \pi_2, ..., \pi_I)$. The conditional density function of γ_i's given $\vec{\pi}$ is:

$$f_{\gamma|\pi}(u) = \pi_1 \cdot \frac{\mathbb{1}_{[a_0,a_1)}(u)}{a_1 - a_0} + ... + \pi_I \cdot \frac{\mathbb{1}_{[a_{I-1},a_I]}(u)}{a_I - a_{I-1}},$$

where $\mathbb{1}_{[a_0,a_1)}(u)$ is the indicator function, receiving the value 1 if u is in the interval $[a_0, a_1)$. In order to estimate the parameter vector $\vec{\pi}$, we assume a Polya tree prior: we assume $I-1$ Beta random variables are positioned in a binary tree-like hierarchy, and that the values of $\vec{\pi}$ are formed by multiplying the values Beta random variables in the paths formed by the tree, as explained in Figure 1. As a default choice we assume all Beta random variables in the Polya tree are $Beta(1,1)$, however, other hyper-parameter choices are also supported by the package, see additional details in Section 6.

The function $f_{\gamma|\pi}$ is estimated by sampling vectors from the posterior distribution of $\vec{\pi}|\vec{X}$ using an MCMC algorithm breifly described in Section 1.1. We finish this section with two important notes:

(1) It is assumed that N_i and P_i are independent.

(2) The `mcleod` package also supports a second type of prior for $\vec{\pi}$, called a Dirichlet tree prior. The Dirichlet tree prior assumes the values of π are formed by a two layer tree with the following structure: the root of the tree is associated with a Dirichlet random variable with I_1 components. Each of the components correspond to with a child node (in the second layer of the tree), which in turn, is associated with a Dirichlet random variable with I_2 components. Overall, the tree has $I = I_1 \cdot I_2$ leaves, associated

with I entries in the vector $\vec{\pi}$. The probability vector $\vec{\pi}$ for this model is formed by multipling the values of Dirichlet vector components, as descending down the tree, in manner similar to Figure 1. The MCMC algorithm for estimating $\vec{\pi}$ for this type of prior is similar to the one presented in Section 1.1.

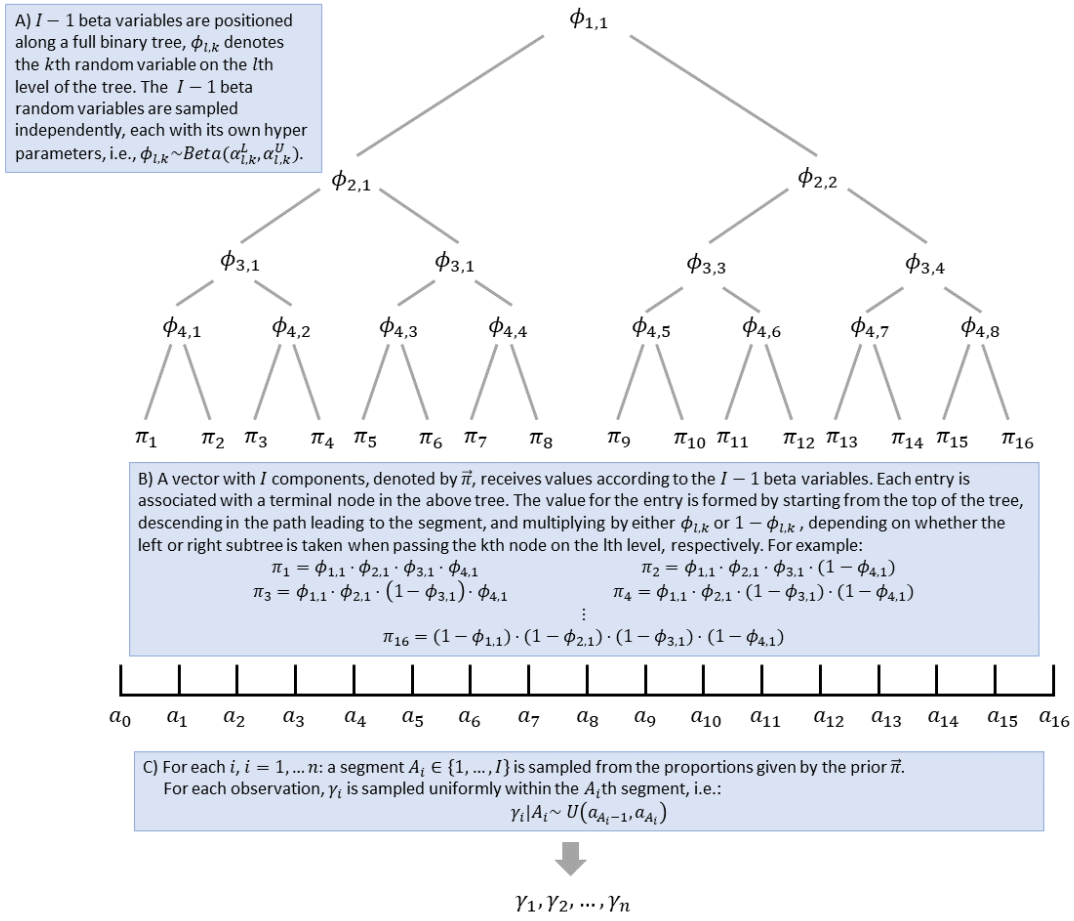


Figure 1: Generative model of the γ_i's

2.1 Sampling from the posterior distribution

The MCMC algorithm used for sampling from the posterior distribution of $\vec{\pi}$ can be briefly described as follows: (1) Given $\vec{\pi} = \vec{v}$, one can draw a sample from the posterior distribution of $\gamma_i | X_i = x_i, \vec{\pi} = \vec{v}$. (2) Similarly, given the values of γ_i's, one can draw of sample from the posterior distribution of $\vec{\pi} | \vec{\gamma}, \vec{X}$, where $\vec{\gamma}, \vec{X}$ are the samples for γ_i's and X_i's, respectively. (3) In the ith iteration, the MCMC algorithm samples values for the γ_i's (using (1), and based on the algorithm current idea of $\vec{\pi}$). Next, using the γ_i's sampled (and the posterior distribution of (2)), the algorithm samples a new value for $\vec{\pi}$. This process is performed iteratively, in order to receive an MCMC set of samples for both $\vec{\pi}$ and the γ_i's.

In order to obtain an estimator for $\vec{\pi}$ (and $f_{\gamma|\pi}$), we run the MCMC sampler for a large number of iterations, e.g., 500 or 1000, and estimate $\vec{\pi}$ using the empirical mean of the obtained samples. "Burn-in" samples from the start of the chain, usually 200-300 sampels, are disregarded when computing the estimate for $\vec{\pi}$.

3 Deconvolution estimates

In this section we present code examples showing how to estimate the mixing distribution in various cases.

3.1 Deconvolution for binomial errors

We generate 300 samples, with $X_i \sim bin(30, P_i)$, and $log(P_i/(1-P_i))$ sampled from a mixture $0.5 \cdot N(-1, 0.3^2) + 0.5 \cdot N(2, 0.3^2)$.

```
N = 30
K = 300
set.seed(1)
u = sample(c(0,1),size = K,replace = T)
x = rbinom(K,size = N,prob =inv.log.odds(rnorm(K,-1+3*u,sd = 0.3)))
n = rep(N,K)

#%%% Peak at data:
head(cbind(x,n))
```

```
##       x  n
## [1,] 11 30
## [2,] 25 30
## [3,]  6 30
## [4,]  5 30
## [5,] 26 30
## [6,]  3 30
```

The simplest was to obtain deconvolution estimates is to run the function `mcleod`, receiving the X_i's and N_i's:

```
res = mcleod(x, n)
```

The posterior samples for $\vec{\pi}$ are available under `res$additional$original_stat_res$pi_smp`. The matrix has I rows, corresponding to the segments of the prior distribution, and columns corresponding the MCMC iterations. For example, the CDF for the 100th MCMC sample for $\vec{\pi}$ can be obtained via:

```
iter_number = 100
res$additional$original_stat_res$pi_smp[,iter_number]
```

We can obtain both the posterior mean density and the posterior mean of the CDF using the function `mcleod.get.posterior.mixing.dist`:

```
posterior_mixing_dist = mcleod.get.posterior.mixing.dist(res)
```

The grid $\vec{a}$ can be retrieved using the entry `a.vec`:

```
head(posterior_mixing_dist$a.vec)
```

```
## [1] -4.000 -3.875 -3.750 -3.625 -3.500 -3.375
```

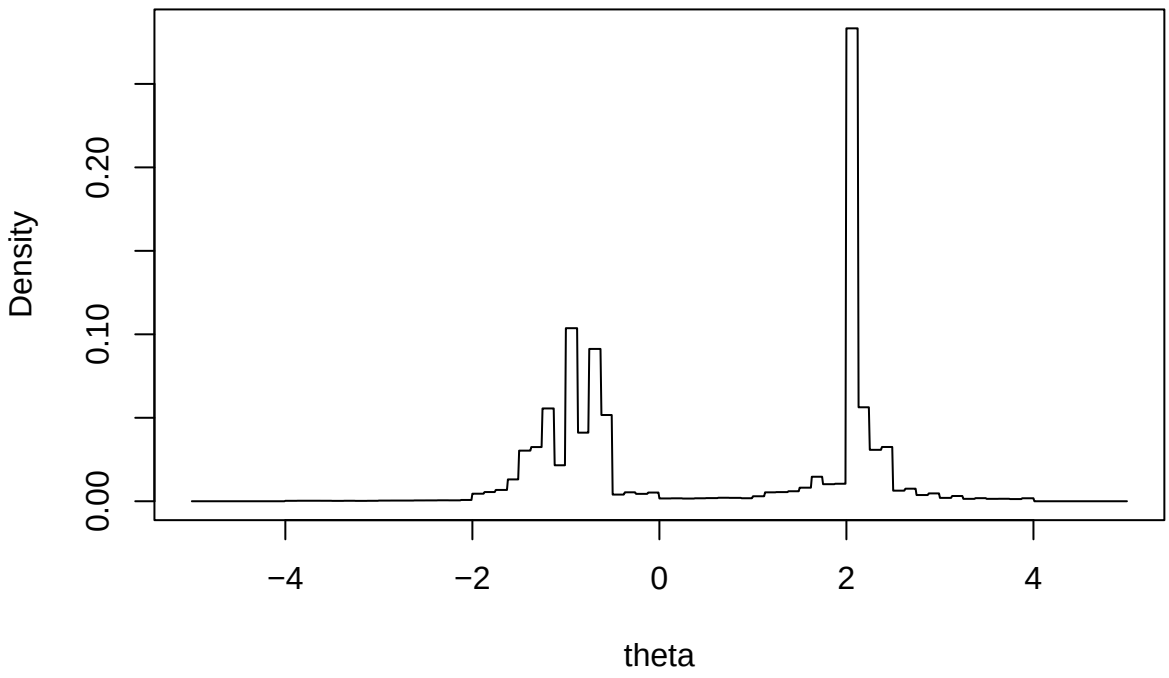
The posterior mean of $\vec{\pi}$ is available under `pi_smp`:

```
head(posterior_mixing_dist$pi_smp)
```

```
## [1] 0.0002519645 0.0002993288 0.0003022684 0.0002759235 0.0002225758
## [6] 0.0002825859
```

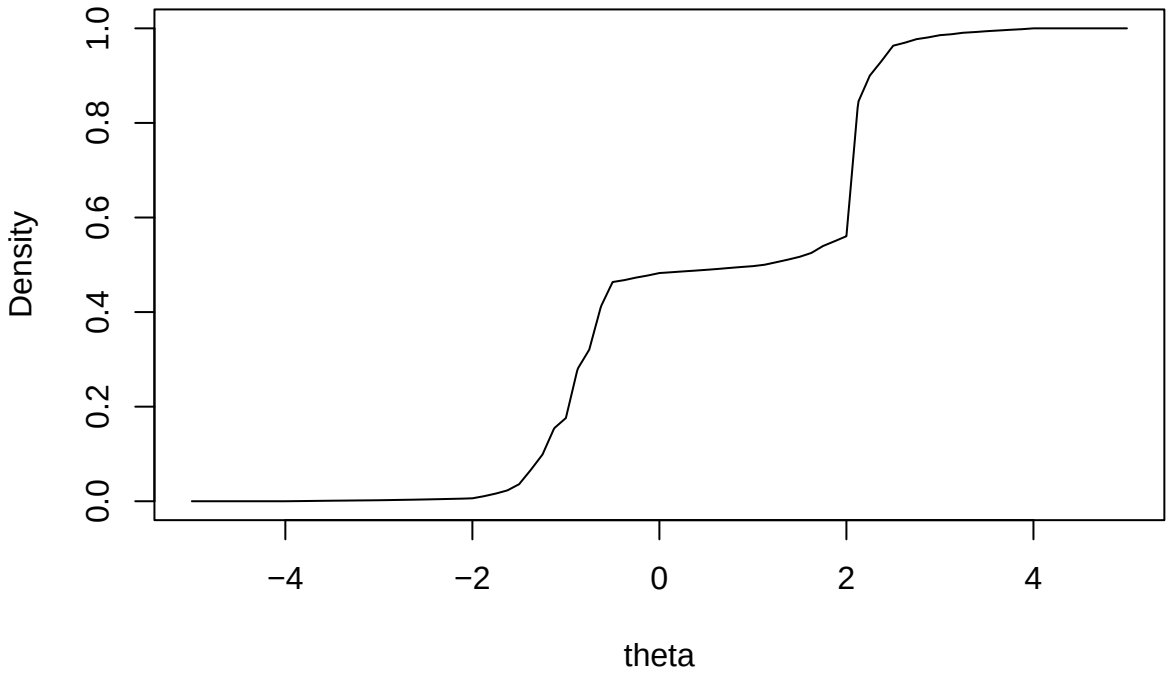
See the documentation for `mcleod.get.posterior.mixing.dist` for an explanation on how to obtain the component-wise posterior median of $\vec{\pi}$. The density for the posterior mean can be obtained via the entry `density`, which is an `R`function. The next code snippet shows how to plot the (pointwise, across log-odds) posterior mean of the density function:

```
x_points = seq(-5,5,0.01)
plot(x_points, posterior_mixing_dist$density(x_points),
     type = 'l',xlab = 'theta',ylab = 'Density')
```



Similarly, we also have the estimated CDF, obtained ia the entry named `CDF`. We plot the estimate for the CDF in the next code snippet:

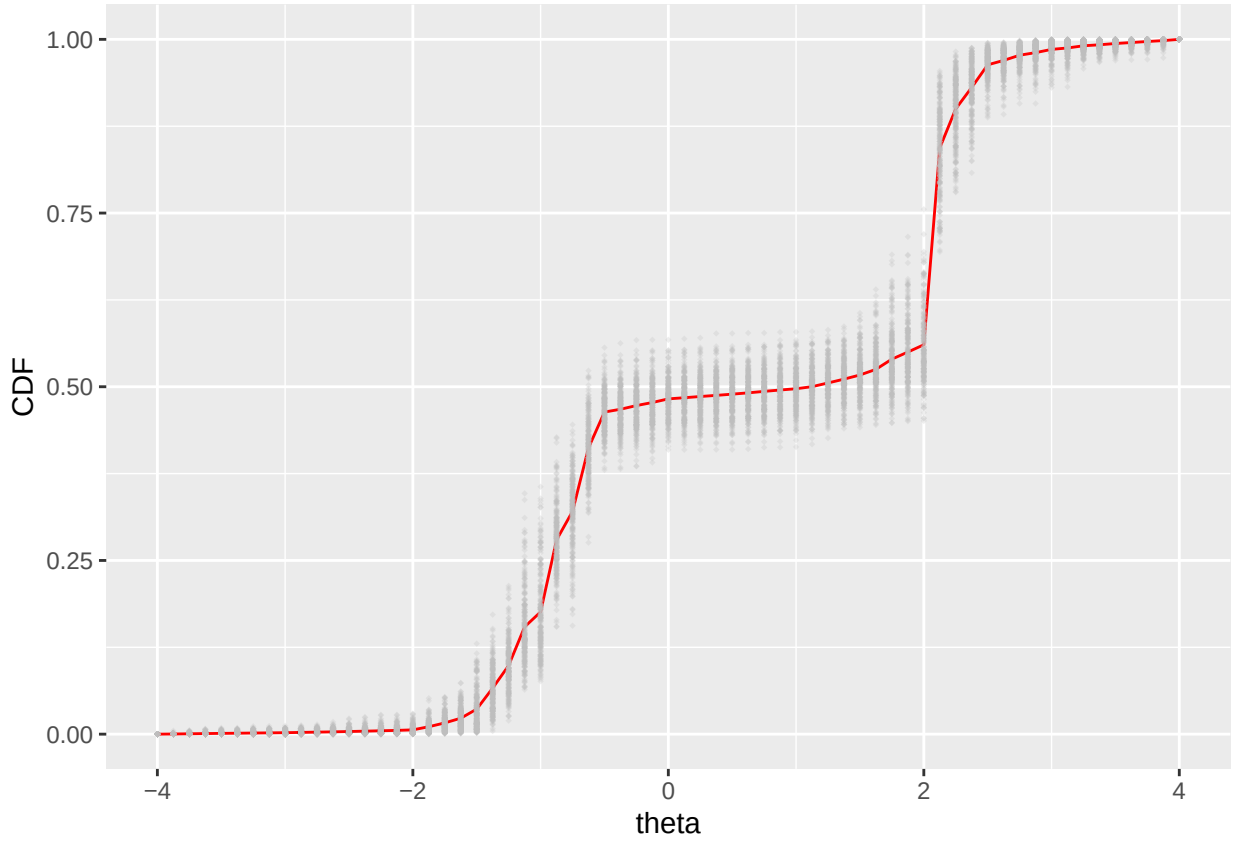
```
x_points = seq(-5,5,0.01)
plot(x_points, posterior_mixing_dist$CDF(x_points),
     type = 'l',xlab = 'theta',ylab = 'Density')
```



We can plot the posterior means of the CDF of the mixing distribution, along with a point cloud for the distribution of samples, using the following command:

```
plot.posterior(res)
```

```
## Warning: Removed 14 rows containing missing values (geom_point).
```



3.1.1 Changing model parameters

We proceed to show how various parameters can be changed. We can change the limits for $\vec{a}$ using the argument `a.limits`:

```
res = mcleod(x, n, a.limits = c(-5,5))
```

We can change the prior and hyperparameters for the mixing distribution by constructing an object using `mcleod.prior.parameters`, and passing it as an argument to `prior_parameters`. For example, here we consruct a finer estimate of the mixing distribution, using a tree with 6 layers.

```
prior_obj  = mcleod.prior.parameters( #construct object defining the prior
  prior.type =MCLEOD.PRIOR.TYPE.BETA.HEIRARCHICAL, #type of prior
  Beta.Heirarchical.Levels = 6
  )

res = mcleod(x, n, prior_parameters = prior_obj) #pass object to main function as argument
```

Another type of prior is the two layer dirichlet tree. This prior requires the number of intervals (corresponding to leaves), and the number of nodes in the first layer.The prior can be defined using the following function:

```
prior_obj  = mcleod.prior.parameters(
  prior.type =MCLEOD.PRIOR.TYPE.TWO.LAYER.DIRICHLET, # define a 2-layer Dirichlet tree
  Two.Layer.Dirichlet.Intervals = 64, #number of segments
  # Note: Two.Layer.Dirichlet.Intervals must
  # be an integer multiple of
  #Two.Layer.Dirichlet.Nodes.in.First.Layer
  Two.Layer.Dirichlet.Nodes.in.First.Layer = 8
```

```
  )

res = mcleod(x, n, prior_parameters = prior_obj) #pass as argument
```

We can change the number of MCMC iterations, together with the number of iterations taken as "burn-in" period, using the function `mcleod.computational.parameters`:

```
comp_obj = mcleod.computational.parameters(nr.gibbs = 500, #define the number of iter.s
                                           nr.gibbs.burnin = 250)

res = mcleod(x, n, computational_parameters = comp_obj) # pass object as argument
```

3.1.2 Posterior estimates for P_i

After fitting a `mcleod model` for the data, we can obtain posterior estimates for P_i given X_i and the estimate for the mixing distribution. This can be done both for the training data, and for out-of-sample observations. The function `mcleod.posterior.estimates.random.effect` is called with the values of X_i, N_i and the fitted model. The next code snippet shows how to compute the posterior mean for $log(P_i/(1-P_i))$, for two observations: one with $X_1 = 5, N_1 = 30$, and one with $X_2 = 35, N_2 = 40$.

```
estimated.log.odds = mcleod.posterior.estimates.random.effect(
  #A vector, giving for each observation the number of successful draws
  X = c(5,35),
  #A vector, giving for each observation the total number of draws
  N = c(30,40),
  #the fitted model for the mixing distribution
  res)

#%% The estimated value for log(P_i/(1-P_i)):
estimated.log.odds
```

```
## [1] -1.027653  2.372869
```

We can convert the estimated log-odds to the probability scale using `inv.log.odds`:

```
inv.log.odds(estimated.log.odds)
```

```
## [1] 0.2635394 0.9147349
```

Note that we can run the function `mcleod.posterior.estimates.random.effect` with the argument `method='mode'`, in order to compute the posterior mode estimates for $log(P_i/(1-P_i))$

3.1.3 Predictive intervals for X_i

We can compute predictive intervals for X_i, for out-of-sample observations, using the function `mcleod.predictive.interval`. The function receives the values of N_i's for the out-of sample observations, and returns the lower and upper bounds. For example, in order to compute 95% predictive intervals for two observations, with $N_1 = 30, N_2 = 60$, we run:

```
mcleod.predictive.interval(
  N = c(30,60),
  mcleod_res = res,
  Interval.Coverage = 0.95)
```

```
## $Lower
## [1] 4 8
##
## $Upper
```

```
## [1] 29 58
```

The returned object contains two vectors, named `Lower` and `Upper`, containing the lower and upper ends of the predictive intervals, corresponding to the observations defined by `N`.

3.2 Deconvolution for Poisson errors

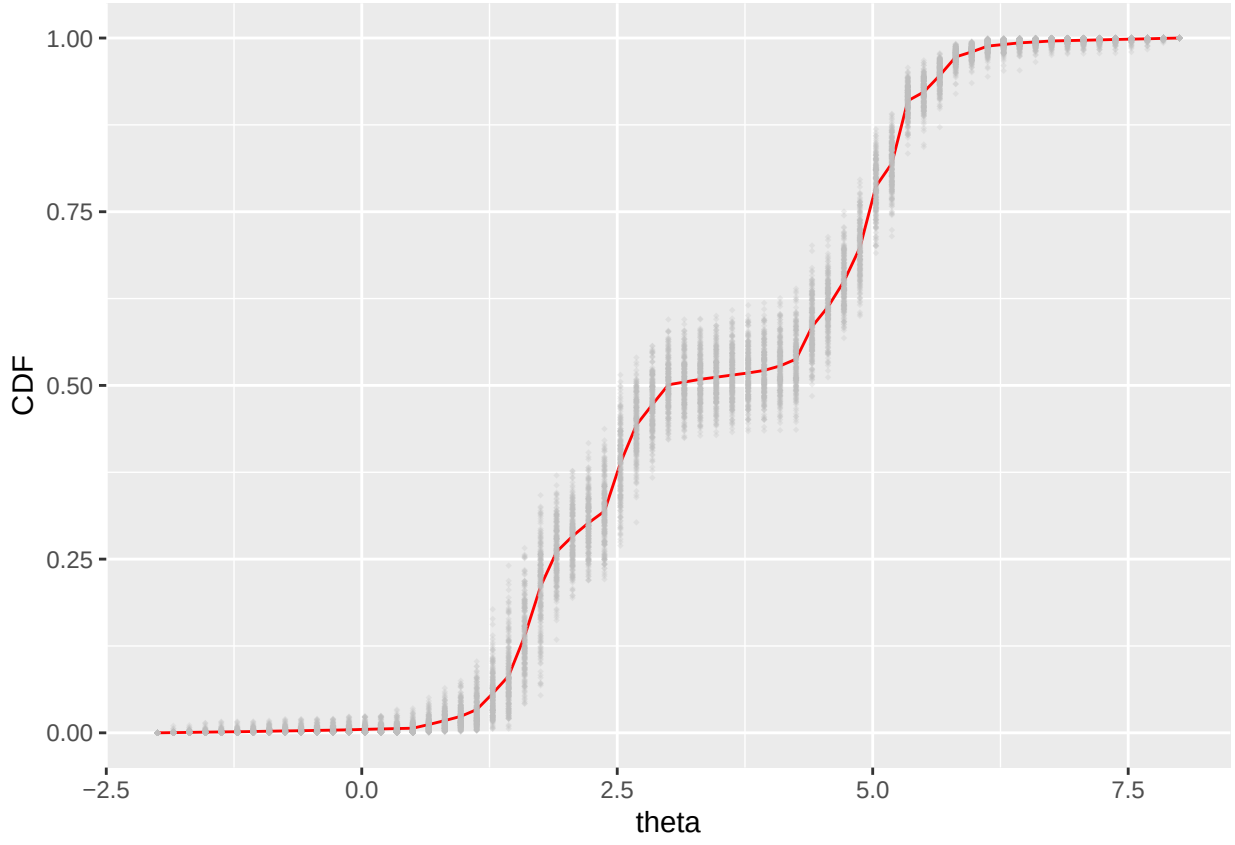
We show to estimate the mixing distribution when the data is sampled using a Poisson distribution. In our example $\log(\lambda_i)$ is distributed via a mixture of two normal components: $0.5 * N(2, 0.5^2) + 0.5 * N(5, 0.5^2)$.

```
K = 200 # number of samples
set.seed(1)
#u sets right or left component in the mix. dist. for each obs.:
u = sample(c(0,1),size = K,replace = T)
x = rpois(K,lambda = exp(rnorm(K,2 + 3*u,0.5)) ) #sample the obs
```

For Poisson distributed data, we run the `mcleod` function with `n.smp` set to `NULL`, and setting `Noise_Type` to `MCLEOD.POISSON.ERRORS`. Other than that, parameters may be passed to `computational_parameters` and `prior_parameters` as in the previous examples. The following example also plots the posterior distribution.

```
res = mcleod(x, n.smp = NULL,a.limits = c(-2,8),Noise_Type = MCLEOD.POISSON.ERRORS)

plot.posterior(res)
```

```
## Warning: Removed 11 rows containing missing values (geom_point).
```



3.2.1 Posterior estimates for λ_i

Posterior estimates for $log(\lambda_i)$ can be obtained via `mcleod.posterior.estimates.random.effect`, as for binomial data. The next code snippet obtains the posterior mean for $log(\lambda_i)$, for observations with $X_1 = 10$ and $X_2 = 200$. Note the `N` is set to `NULL`.

```
estimated.log.lambda = mcleod.posterior.estimates.random.effect(
  #Number of obs. counts per Poisson sample
  X = c(10,200),
  #Set N to NULL
  N = NULL,
  #the fitted model for the mixing distribution
  res)

#%% The estimated value for log(lambda_i):
estimated.log.lambda
```

```
## [1] 2.491048 5.307971
```

The estimated value can be converted back to the $lambda_i$ scale by exponentiation:

```
exp(estimated.log.lambda)
```

```
## [1]  12.07393 201.94003
```

3.2.2 Predictive intervals for X_i

A predictive interval for X_i can be obtained via `mcleod.predictive.interval`, as for binomial data. Note that since N_i is not defined for the data, all observations are i.i.d., and we have a single predictive interval:

```
mcleod.predictive.interval(
  N = NULL,
  mcleod_res = res,
  Interval.Coverage = 0.95)
```

```
## $Lower
## [1] 1
##
## $Upper
## [1] 308.175
```

4 Constructing confidence intervals for the CDF of rate parameters

This section explains how to construct point-wise confidence intervals for the CDF of the posterior distribution of P_i's, given the data. We begin by briefly explaining the method, and then proceed to show code snippets explaining how the computation is done in practice.

4.1 Method

Let $CDF(p)$ be the CDF value for the mixing distribution at point p, $p \in (0, 1)$. In order to construct CIs for $CDF(p)$, we invert test for the following types of hypotheses:

$$H_0^{LE}(\tilde{q}, \tilde{q}) : \tilde{q} \leq CDF(\tilde{p})$$

$$H_0^{GE}(\tilde{q}, \tilde{p}) : CDF(\tilde{p}) \leq \tilde{q}$$

Specifically, when constructing a two-sided confidence interval at level $1-\alpha$ for $CDF(\tilde{p})$, in order to find the lower end of the CI, we find the the highest $\tilde{q}$ for which we do not reject the null hypothesis $H_0^{GE}(\tilde{q},\tilde{p})$ at level $\alpha/2$. Similarly, the upper end of the CI is found by the finding the lowest $\tilde{q}$ for which we do not reject $H_0^{LE}(\tilde{q},\tilde{p})$ at level $\alpha/2$.

We proceed to explain how the hypotheses $H_0^{GE}(\tilde{q},\tilde{p})$ and $H_0^{LE}(\tilde{q},\tilde{p})$ are tested. Since the procedures for the two hypotheses are similar, we describe the procedure for $H_0^{GE}(\tilde{q},\tilde{p})$ alone. Throughout the explanation we make use of Figure~2. Let $\hat{CDF}(p)$ denote an estimator for $CDF(p)$, computed using the point-wise median at each value of p. This estimator is represented in Figure 2 by the blue curve. In order to test $H_0^{GE}(q,p)$, we compare the function $\hat{CDF}(p)$ for the data with similar deconvolution estimates, obtained for a distribution for which the null hypothesis. Specifically, our valid test for $H_0^{GE}(\tilde{q},\tilde{p})$ would compare $\hat{CDF}$ with estimates obtained from data samples, generated from the sthocastically smallest distribution for which $H_0^{GE}(\tilde{q},\tilde{p})$ holds. This distribution has $\tilde{q}$ mass at $p=0$ and a mass of $1-\tilde{q}$ at $p=\tilde{p}$, and is represented by the orange curve in Figure 2.

The step for testing $H_0^{GE}(\tilde{q},\tilde{p})$ are as follows: (1) Estimate $\hat{CDF}(p)$ for the data.

(2) Sample B data sets of the form $X_i \sim bin(N_i, P_i), i \in \{1,...,n\}$, where the N_i's are identical to the values for the observed data, and the P_i's are sampled independently from a distribution with $\tilde{q}$ mass at $p=0$ and a mass of $1-\tilde{q}$ at $p=\tilde{p}$. For each of the B data generations, estimate the mixing distribution. Let $\hat{CDF}^{(b)}(p), b \in \{1,...,b\}$ denote the estimated CDFs for the observed dataset. A graphical depiction of the estimated CDFs for the generated samples is shown using the green curves in Figure 2.

(3) Compute a P-value using the following formula:

$$PV = \frac{1+\sum_{b=1}^{B} I\left(\hat{CDF}(\tilde{p}-\rho) \leq \hat{CDF}^{(b)}(\tilde{p}-\rho)\right)}{1+B},$$

where ρ is a parameter of the method, see additional figures below. The PV can be interpreted as counting the relative portion of green curves that are found above the blue curve, at point $\tilde{p}-\rho$ (shown in purple in Figure 2). In Figure 2, we have $B=3$ green curves found above the blue curve, so we know the PV is $\frac{1+3}{1+3}$ or 1.

Yekutieli et al. (2022) discuss the importance of the parameter rho: they show that by testing GE hypotheses by taking the values of CDF estimates at $\tilde{p}-\rho$ instead of $\tilde{p}$, the test potentially has higher power, if ρ is calibrated correctly. The authors suggest the following calibratio procedure for ρ: (1) pick, at random, 10%-20% of the samples, and exclude them from the data used for testing; (2) Let $p_1, p_2, ..., p_m$ be a series of values in the range $(0,1)$. In addition, let $\rho_1, \rho_2, ..., \rho_T$ be a set of T candidate values for ρ. Constructing CIs for $CDF(p_1), CDF(p_2), ..., CDF(p_m)$ using each of the T values for rho. Let $LengthCI(p_i, \rho_j)$ denote the length of the CI for $CDF(p_i)$, when statistical testing is done using ρ_j. (3) The authors suggest selecting ρ using:

$$\rho = arg\ min_{\rho \in \{\rho_1, \rho_2, ..., \rho_T\}} \sum_{i=1}^{m} LengthCI(p_i, \rho).$$

The authors suggest selecting the values $p_1, p_2, ..., p_T$ to be the quantiles for which the estimated CDF for mixing distribution in the holdout data obtains the values $0.1, 0.2, ..., 0.9$. In addition, the package also include default values for $\rho_1, \rho_2, ..., \rho_T$. See additional details in the on the calibration method for ρ in the manuscript.

When testing $H_0^{LE}(\tilde{q},\tilde{q})$ hypotheses, there are two differences: (1) bootstrap samples are generated using a mixing distribution with $\tilde{q}$ mass at $p=\tilde{p}$ and a mass of $1-\tilde{q}$ at $p=1$; (2) testing is done at the quantile $\tilde{p}+\rho$; and (3) Pvalues are computed using a left tailed rejection region, i.e., by checking the proportion of $\hat{CDF}^{(b)}(\tilde{p}+\rho)$'s smaller than $\hat{CDF}(\tilde{p}+\rho)$.

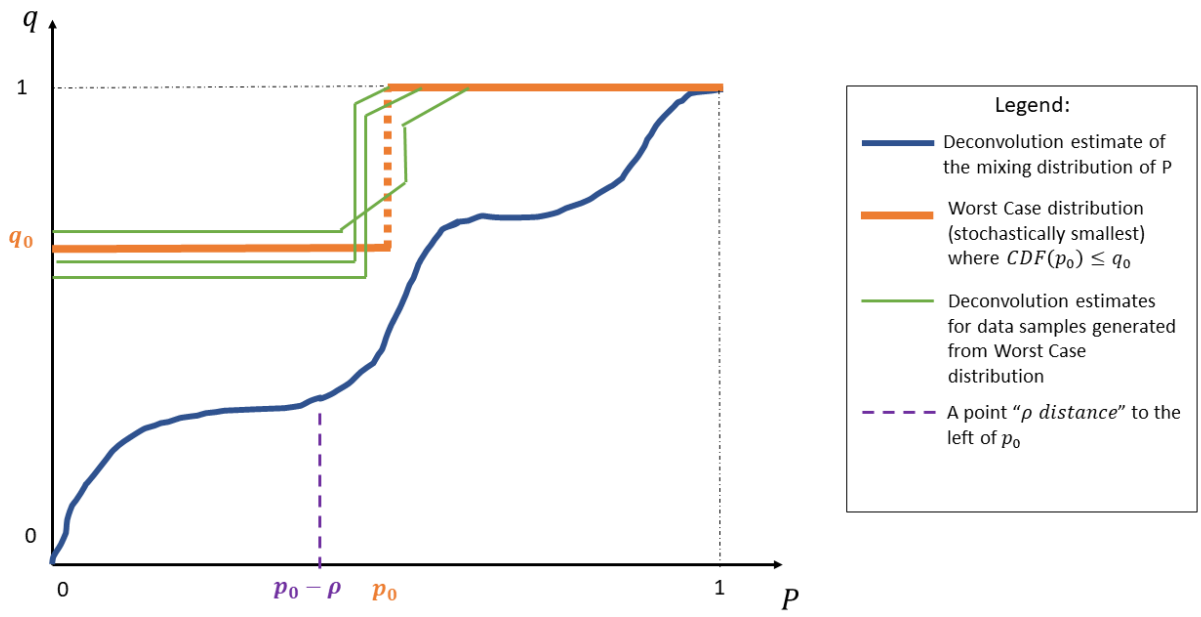


Figure 2: Graphical representation of a test for $H_0^{GE}(q, p)$

4.1.1 Speeding up computations

The above method is computationally intensive, however there are several steps taken to speed up computation: * Computation of P-values is done parallely, using `R` package `doRNG`. The number of cores used can be set by the user.

- When computing P-values, if the first $\alpha \cdot B$ permutations are more extreme than oberserved test statistic, the computation is halted and a PV of 1 is returned.
- Let p, p' be two quantiles of the mixing distribution, with $p < p'$. Due to the monotonicty of the CDF of the mixing distribution, it is unknown that the lower end of the CI for $CDF(p')$ is greater than or equal to the lower end of the CI for $CDF(p)$. The upper ends of the CIs share the same relationship. Therefore, if $H_0^{GE}(q, p)$ was rejected at significance level α, we need not test $H_0^{GE}(q, p')$, and can proceed directly to testing CDF values higher than q.
- The code only needs to generate bootstrap samples for $H_0^{GE}(q, p)$. Bootstrap samples for $H_0^{LE}(q, p)$ are obtained without additional computation using the following "hack": Let $\theta = log(p/(1-p))$ denote the log-odds for p. Let $C\hat{D}F(\theta)$ denote a CDF estimate parameterized using the parameter *theta* (i.e., $C\hat{D}F\left(\frac{exp(p)}{1+exp(p)}\right)$). If $C\hat{D}F(\theta)$ is a deconvolution estimate for a sample from the worst case (stochastically lowest) distribution of $H_0^{GE}(q, p)$, then $C\hat{D}F(-\theta)$ is equivalent to a deconvolution estimate for a sample from the worst case (stochastically highest) distribution of $H_0^{GE}(1-q, 1-p)$. See additional details in paper and the internal function `mcleod:::mcleod.CI.deconv.bank.get_median_curves_for_worst_case_hypothesis_at_point`.
- We testing $H_0^{GE}(q, p)$, a lower bound for the P-value is given by $F_{bin(n,q)}(n \cdot C\hat{D}F(p-\rho))$, where $F_{bin(n,q)}$ is the CDF of a binomial random variable with n draws and probability q for success in each draw. Similarly, for $H_0^{LE}(q, p)$, a lower bound for the P-value is given by $F_{bin(n,1-q)}(n \cdot (1 - C\hat{D}F(p-\rho)))$. The code checks if the lower bound for the P-value is greater than $\alpha/2$. If this is the case, we do not need to perform permutations. See additional details on how this bound was derived in the paper.

4.2 Pointwise Confidence intervals for CDF, binomial noise

For the example, we 500 binomially distributed samples with $X_i \sim bin(20, P_i)$, and $log(P_i/(1-P_i))$ distributed according to a bi-normal mixture: $log(P_i/(1-P_i)) \sim 0.5 \cdot N(-2, 0.5^2) + 0.5 \cdot N(-1, 0.3^2)$.

```
n = 500
N = rep(20,n)
  set.seed(1)
p = inv.log.odds(rnorm(n,-2,0.5)+3*rbinom(n,1,0.3))

X = rbinom(n = n,size = N,prob = p)

 # we obtain the true distribution $P_i$'s by sampling 10^6 observations,
 # later used for plottin
p_true = inv.log.odds(rnorm(1E6,-2,0.5)+3*rbinom(1E6,1,0.3))
```

Confidence intervals for the CDF, across all values of $\theta = log(P_i/(1-P_i))$, are estimated via the following functions. The code takes a few minutes to run, and prints progress to screen. This can be changed by setting `verbose` to `F`. For this run, we use all the data for testing, and use a fixed value of $\rho = 0.25$. Calibration of ρ will be discussed in the next examples.

```
CI.est.res = mcleod.estimate.CI(X = X, N = N,
        CI_param = mcleod.CI.estimation.parameters(rho.set.value = 0.25))
```

Next, we plot the estimated CDF in red (posterior median), the pointwise confidence intervals in black, and the true mixing distribution in blue.

```
plot.mcleod.CI(mcleod.CI.obj = CI.est.res) #one command to plots CIs

# plots the oracle:
oracle_x = seq(0.01,0.99,0.01)
lines(oracle_x,(ecdf(p_true))(oracle_x),col = 'blue',lwd =1.5)
```

The above code snippet was run before-hand, and the result is displayed here as a figure:

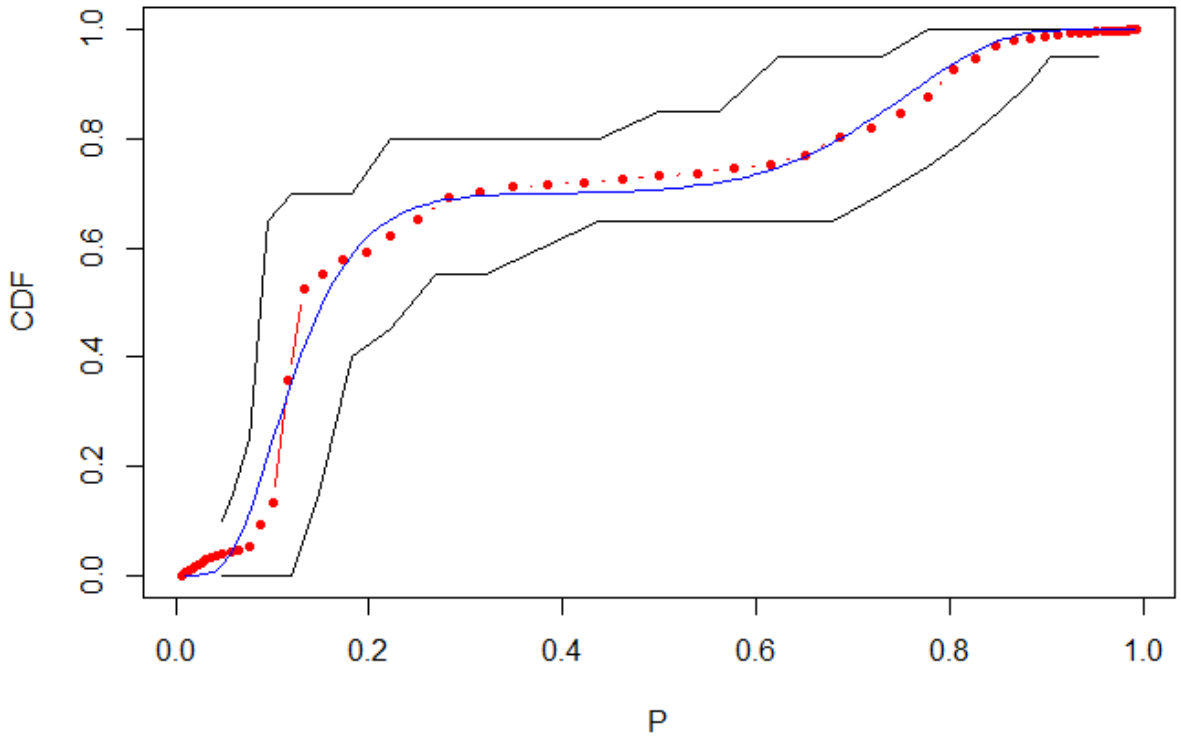


Figure 3: Estimated mixing distribution (pointwise posterior median in red), 95% pointwise CIs (in black) and true mixing distribution (in blue)

The exact values for the CI can be extracted via the function `mcleod.get.CIs.mixing.dist`:

```
dt_CIs = mcleod.get.CIs.mixing.dist(CI.est.res)

dt_CIs
```

We move to a more advanced example, showing how (a) ρ can be estimated from the data, (b) how to refine the grid of quantiles and CDF values, over which CIs are constructed, and (c) how to change to confidence level for the CIs.

```
  # We construct a CI object holding the different definitions
  CI_param = mcleod.CI.estimation.parameters(
    # grid of mixing distribution quantiles. Here we work with 0.1 steps, instead of 0.25
    theta_vec = seq(-4,4,0.1),
    # grid of CDF values. Previous grid has 0.05 jumps
```

```
  q_vec = seq(0.05,0.95,0.025),
  # We change the confidence level to 0.9 (instead of the default 95% CI)
  alpha.CI = 0.9,
  # we consider 0.1,0.2,0.3 as possible values for rho
  rho.possible.values = seq(0.1,0.3,0.1))

# we pass CI_param as an object, and also set the
# relative part of samples used for calibrating rho
# here we pick 10% of the data. Values in the range 0.1-0.2 are reasonable
CI.est.res = mcleod.estimate.CI(X = X,
                                N = N,
                                CI_param = CI_param,
                                ratio_holdout = 0.1)
```

Running the code with the following parameters leads to finer CI's

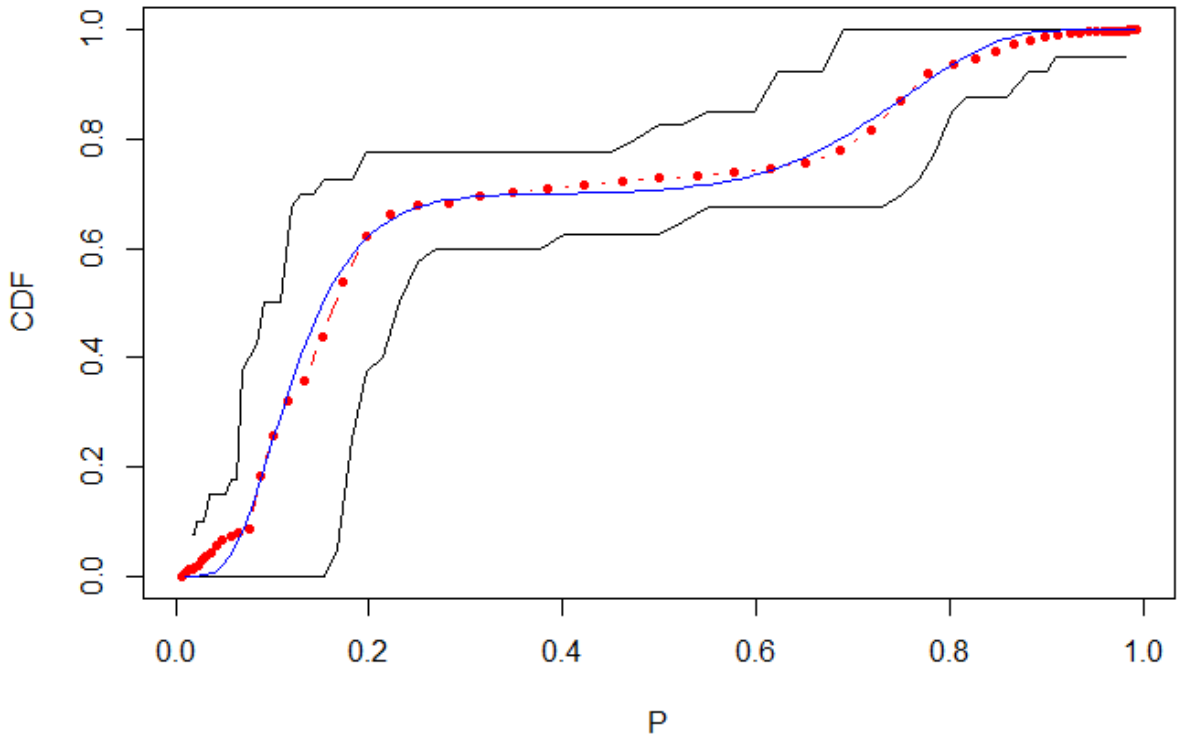


Figure 4: Estimated mixing distribution (pointwise posterior median in red), 95% pointwise CIs (in black) and true mixing distribution (in blue). Estimation done over a finer grid compared to Figure 3.

4.3 Computationally efficient confidence intervals for the quantile of the mixing distribution that corresponds to a given CDF value

The function `mcleod.estimate.CI.single.q` is appropriate for cases where a single confidence interval for a given CDF value is needed. The function receives the data, toghether with a given CDF value, e.g. `q=0.5`

for the median. The function outputs a confidence interval, in terms of quantiles (values of θ) for the given CDF value.

The following code snippet computes a CI for the median of the mixing distribution. In this code snippet, we choose a resolution of 0.05 for the reported confidence interval. Calibration of ρ is done automatically using 10% of the data. Using a fixed value for ρ reduces the running time by about 50%.

```
  set.seed(1)

  CI.res = mcleod.estimate.CI.single.q(X = X,N = N,q = 0.5, #the wanted CDF value
     #set resolution:
   CI_param = mcleod.CI.estimation.parameters(theta_vec = seq(-3,3,0.05),
                                              rho.set.value = 0.25),
            verbose = F) # can ask to print progress. Takes around 2 minutes.

  CI.res
```

4.4 Computationally efficient confidence intervals for the CDF value of the mixing distribution at a given quantile

The function `mcleod.estimate.CI.single.theta` is appropriate for cases where a single confidence interval for a given θ value is needed. The function receives the data, toghether with a quantile, and outputs a confidence interval, in terms of CDF values.

The following code snippet computes a CI for the CDF value of the mixing distribution, at the quantile $\theta = 0.5$. Calibration of ρ is done automatically using 10% of the data. Using a fixed value for ρ reduces the running time by about 50%.

```
  CI.res = mcleod.estimate.CI.single.theta(X = X, N = N,
    # the quantile for which the CI is needed
    theta = 0,
    # resolution for the CI, in terms of CDF values
    CI_param = mcleod.CI.estimation.parameters(q_vec = seq(0.05,0.95,0.01),
                                               rho.set.value = 0.25) )

  CI.res
```

4.5 How to compute point-wise confidence intervals for multiple confidence levels simultanously

Users may need P-values for $H_0^{LE}(\tilde{q}, \tilde{q})$ and $H_0^{GE}(\tilde{q}, \tilde{q})$ across all values of $\tilde{q}$'s and $\tilde{q}$'s. For example, when wanting to derive all point-wise confidence intervals, for all confidence levels, simultanously. The following code snippet explains how to generate P-values for all hypotheses simultanously.

```
#Generate a similar dataset, 200 samples
n = 200
N = rep(20,n)
  set.seed(1)
p = inv.log.odds(rnorm(n,-2,0.5)+3*rbinom(n,1,0.3))
X = rbinom(n = n,size = N,prob = p)

# We call mcleod.estimate.CI again, but
CI.est.res = mcleod.estimate.CI(X = X, N = N,
        # turn this flag to true
        compute_P_values_over_grid = T,
        # and this flag to false
```

```
        compute_CI_curves = F,
        # for this example, we use a fixed value of rho=0.25
        CI_param = mcleod.CI.estimation.parameters(rho.set.value = 0.25),
        # this can be used to print progress
        verbose = T)
```

The Pvalues for all GE/LE hypotheses are found in the following fields of the returned object:

```
# Grid of P-values for GE hypotheses. Rows are CDF values, cols are quantiles.
# See rownames and colnames for details.
CI.est.res$pvalues_grid$GE.pval.grid

# Grid of P-values for LE hypotheses. Rows are CDF values, cols are quantiles.
# See rownames and colnames for details.
CI.est.res$pvalues_grid$LE.pval.grid
```

5 GLMs with a random intercept of a general distribution

This section discusses models of the form $X_i \sim bin(N_i, P_i)$, where P_i follows the hierarchical model $log(P_i/(1-P_i)) = \gamma_i + \vec{\beta}^T \vec{Z}_i$, and where γ_i is sampled from the Polya-tree framework shown in Figure 1, $\vec{\beta}$ is a vector of p slopes, and $\vec{Z}_i$ is a vector of sample covariates (with no intercept term).

In order to introduce covariates into the model, we assume $\vec{\beta}$ is distributed $\vec{\beta} \sim N(\vec{0}, \Sigma)$, where Σ is a diagonal matrix. By default, Σ is set to be the unit matrix. In addition to a multivariate normal prior, our code also supports: (1) non-informative priors for slope coefficients, by setting diagona entries of Σ to negative values: (2) Priors of a general distribution, see example below.

Covariates are incorporated into the MCMC algorithm using the following appraoch. At the first iteration, the algorithm samples values for $\vec{\beta}$, and $\vec{\pi}$. In each iteration, the algorithm first samples values for the γ_i's given $\vec{\pi}$ and $\vec{\delta}$. Afterwards the algorithm samples values for $\vec{\pi}$ and $\vec{\beta}$, given the value of the γ_i's, to be used in the next iteration. The value for $\vec{\pi}$ is sampled using a Gibbs sampling step, as in the setting with no covariates. The value of $\vec{\beta}$ is sampled using a Metropolis-Hastings step and the proposal distribution $N(\vec{\beta}', \Sigma_{Proposal})$, where $\vec{\beta}'$ is the vector of slopes for the current iteration, and $\Sigma_{Proposal}$ is a diagonal matrix (the default matrix has the value 0.05 across all diagonal entries). The coefficients vector $\vec{\beta}$ is initialized using a normal random intercept regression model.

In additional to the Binomial model, the package also supports a Poisson regression model of the form $X_i \sim Pois(\lambda_i), log(\lambda_i) \sim \gamma_i + \vec{\beta}^T \vec{Z}_i$, see example below. In addition, our code supports offset terms for the linear predictor (covariates with a slope fixed to 1).

The model assumes $N_i, \vec{Z}_i$, and γ_i are jointly independent.

5.1 Binomial regression with a general random intercept

We show two examples for Binomial regression with a random intercept

5.1.1 Example 1: deafult parameters

The first example has two normally distributed covariates, with slopes being $\vec{\beta} = (-1, 1)$. The γ_i's are distributed Cauchy with a scale of 0.5. The following code snippet generates the data:

```
N = 30 #Number of draws per binomial observations
K = 200 #Number of samples
set.seed(1)
covariates = matrix(rnorm(K*2,sd = 0.5),nrow = K) #Generate covariates
colnames(covariates) = c('covariate 1','covariate 2')
```

```
#define slopes:
real_beta_1 = -1
real_beta_2 = 1
#sample
x = rbinom(K,size = N,
           prob = inv.log.odds(rcauchy(K,location = 0,scale = 0.5) +
                     real_beta_1*covariates[,1] + real_beta_2*covariates[,2]))
n = rep(N,K)

#%%%%%%%%%%%%%%%%%%%%%%%%
#Present the data:
#%%%%%%%%%%%%%%%%%%%%%%%%
head(round(cbind(X = x,N = n,covariates),digits = 2))
```

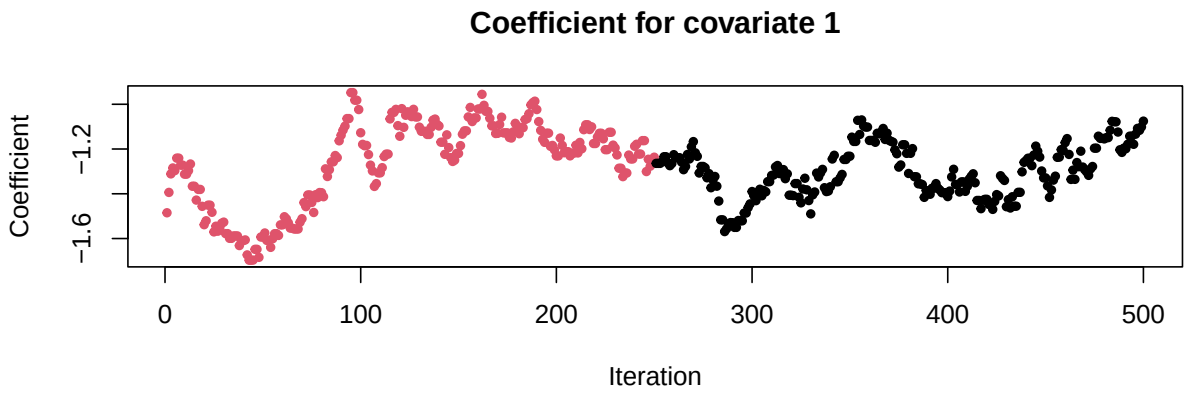
```
##       X  N covariate 1 covariate 2
## [1,] 17 30       -0.31        0.20
## [2,] 20 30        0.09        0.84
## [3,] 24 30       -0.42        0.79
## [4,]  9 30        0.80       -0.17
## [5,]  7 30        0.16       -1.14
## [6,] 25 30       -0.41        1.25
```

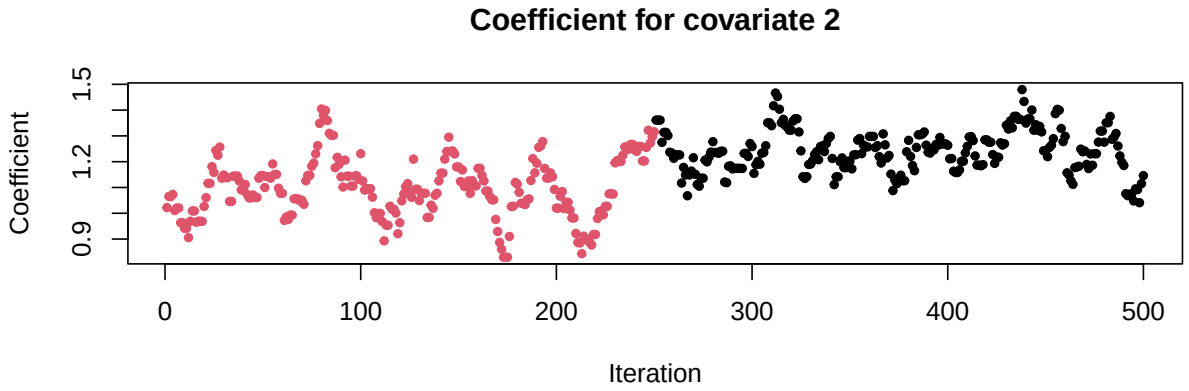
In order to incorporate covariates in the model, pass the matrix of covariates to the argument named `covariates`. Since we set Σ to be the unit matrix, it is best if the covariates are normalized. The model is fitted using the following command:

```
res = mcleod(x, n, covariates = covariates)
```

The function `results.covariate.coefficients.posterior` is used to plot the values sampled for $\vec{\beta}$ in the MCMC algoritm. Points in red are the "burn-in" period, see details on how to change, below.

```
coeffs = mcleod::results.covariate.coefficients.posterior(res)
```





The returned object includes the posterior means for the slope coefficients, and the acceptance rate for the MH steps, used for sampling the values of $\vec{\beta}$. See the documentation for `results.covariate.coefficients.posterior` for information on additional graphs, used for diagnostics of the proposal distribution and acceptance rate.

The posterior means for $\vec{\beta}$:

```
coeffs$posterior.means
```

```
## [1] -1.301947  1.241028
```

The mean acceptance rate for Metropolis Hastings steps:

```
coeffs$acceptance.rate
```

```
## [1] 0.81
```

The MCMC samples for $\vec{\beta}$:

```
head(t(coeffs$beta_smp))
```

```
##           [,1]     [,2]
## [1,] -1.484912 1.021976
## [2,] -1.393941 1.065163
## [3,] -1.311140 1.064845
## [4,] -1.284659 1.073135
## [5,] -1.296496 1.010694
## [6,] -1.240732 1.020474
```

The suggested values for $\vec{\beta}$, sampled for MH iterations:

```
head(t(coeffs$beta_suggestion))
```

```
##           [,1]      [,2]
## [1,] -1.393941 1.0651635
## [2,] -1.311140 1.0648450
## [3,] -1.284659 1.0731346
## [4,] -1.296496 1.0106944
## [5,] -1.240732 1.0204742
## [6,] -1.242102 0.9766205
```

For each iteration, was the suggested value for $\vec{\beta}$ accepted (1) or rejected (0):

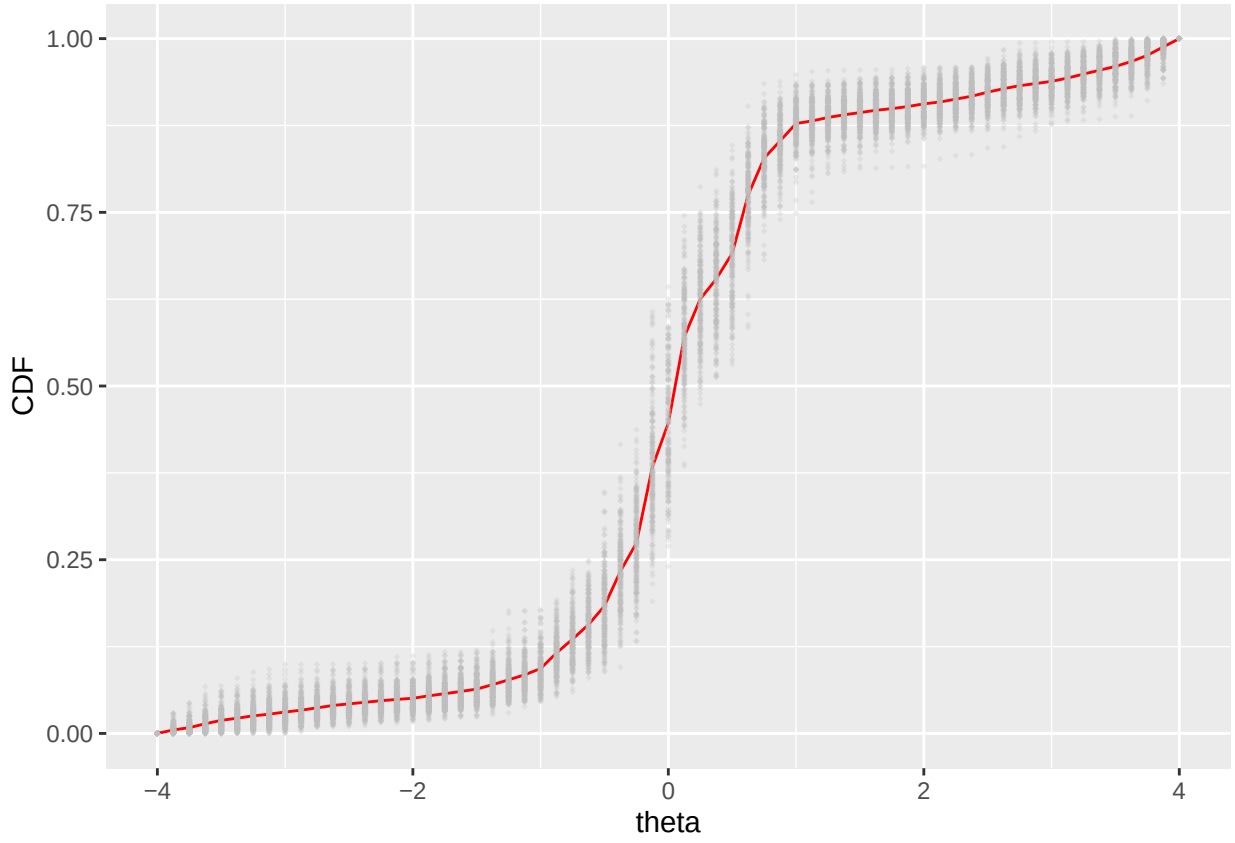
```
head(coeffs$proposal_approved)
```

```
## [1] 1 1 1 1 1 0
```

The posterior distribution of $f_{\gamma_i|\vec{\pi}}$ can still plotted via:

```
plot.posterior(res)
```

```
## Warning: Removed 17 rows containing missing values (geom_point).
```



5.1.2 Posterior estimates for γ_i

Posterior estimates for γ_i can be obtained using `mcleod.posterior.estimates.random.effect` as for the case with no covariates. The code snippet below estimates the values of γ_i for two observations: The first observation is $X_1 = 15, N_1 = 30, \vec{Z}_1 = (0, 0)$; and the second observation is $X_2 = 40, N_2 = 50, \vec{Z}_2 = (-0.5, 0.5)$.

```
estimated.gamma_is = mcleod.posterior.estimates.random.effect(
  #A vector, giving for each observation the number of successful draws
  X = c(15,40),
  #A vector, giving for each observation the total number of draws
  N = c(30,50),
  #the fitted model for the mixing distribution
  mcleod_res = res,
  #Covariates matrix
  covariates = rbind(c(   0,  0),
                     c(-0.5,0.5))
  )

#%% The estimated value for gamma_i's:
estimated.gamma_is
```

```
## [1] 0.2664801 0.3761593
```

5.1.3 Predictive intervals for X_i for a random intercept Binomial regression

Predictive intervals for the value of X_i in out of sample observations can be obtained using `mcleod.predictive.interval`, as for the case with no covariates. The following code snippet generates predictive intervals for two out of sample observations with the following N_i's and covariates: the first observation has $N_1 = 30, \vec{Z}_1 = (-2, 2)$; and the second observation has $N_2 = 50, \vec{Z}_2 = (1, -1)$.

```
mcleod.predictive.interval(
  N = c(30,50),
  mcleod_res = res,
  covariates = rbind(c(   -2,  2),
                     c(  1,-1)),
  Interval.Coverage = 0.95)
```

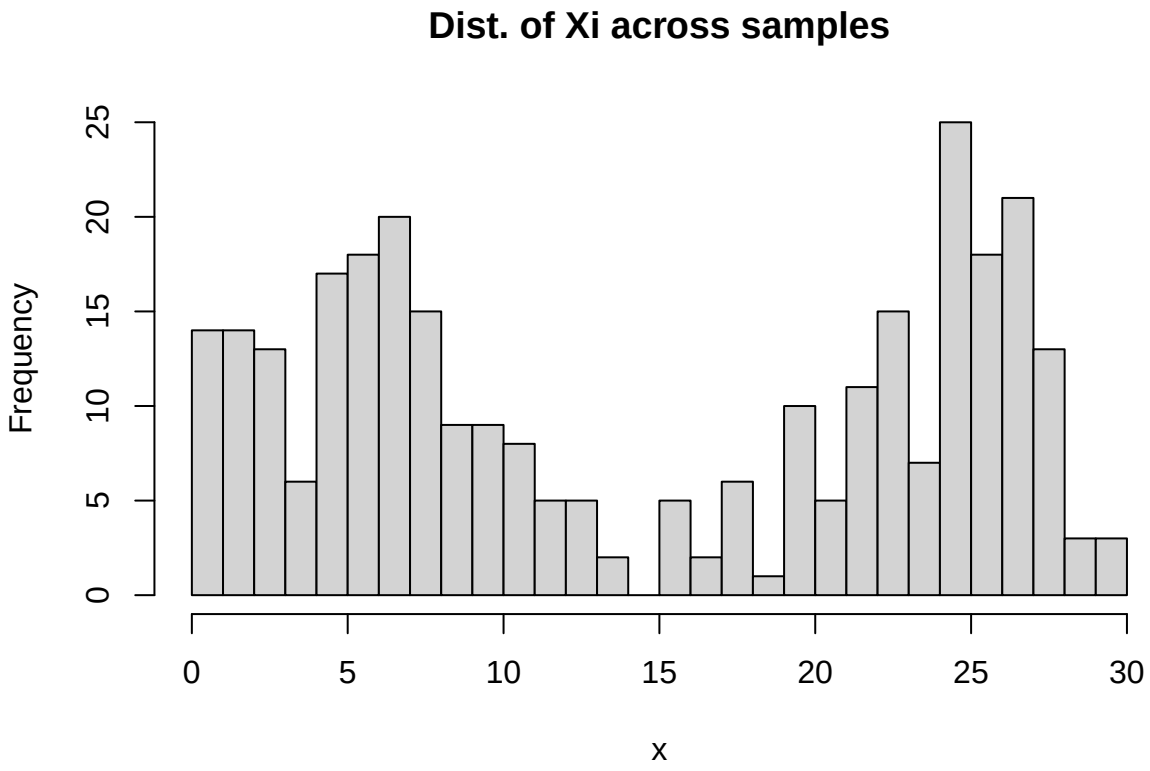
```
## $Lower
## [1] 27  0
##
## $Upper
## [1] 30 39
```

5.1.4 Example 2: How to set different initialization, prior and suggestion distribution for $\vec{\beta}$

By default, the initial MCMC values for $\vec{\beta}$ are set using a random normal intercept binomial regression. However, these values, as well as the prior and proposal distributions for $\vec{\beta}$ may be modified by the user. In the second example, we have a single covariate and a bimodal random intercept distribution. We show how to (a) change the initial values for $\vec{\beta}$, (b) how to change the prior for $\vec{\beta}$, and (c) how to change the proposal distribution for $\vec{\beta}$. The next code snippet generates the example data:

```
  N = 30 # Number of draws per sample
  K = 300 #Number of samples
  set.seed(2)
  covariates = matrix(rexp(K),nrow = K) # exponentially distributed coefficients
  real_beta = -0.5 #the real value of the coefficient

  u = sample(c(0,1),size = K,replace = T)
  x = rbinom(K,size = N,
             prob =inv.log.odds(rnorm(K,-1+3*u,sd = 0.3) +
                                real_beta*covariates))
```

```r
n = rep(N,K)

###%%%%%%%%%%%%%%%%%%%
### plot the distribution of X
###%%%%%%%%%%%%%%%%%%%
hist(x,main ='Dist. of Xi across samples',breaks = 30)
```

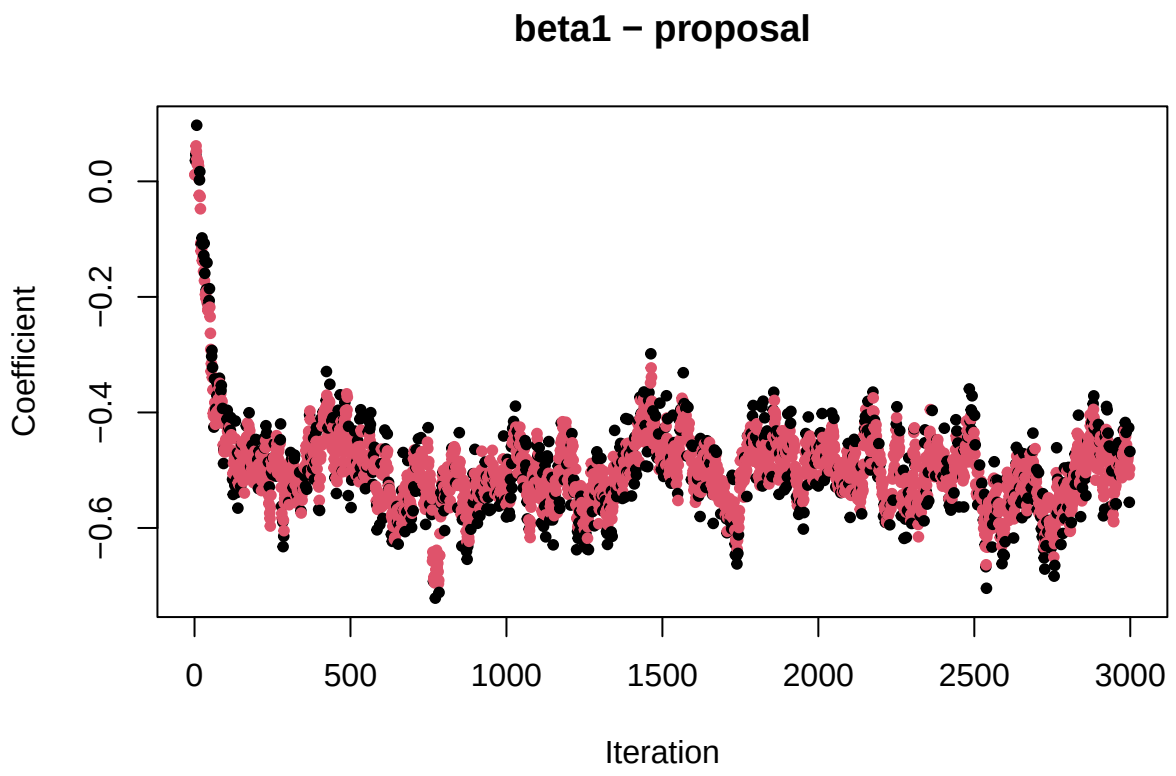


For this example we set the inital values for $\vec{\beta}$ to be 0 (using the argument `beta_init` below). In addition, we set the standard deviationfor the prior of beta to be 1.5 (using the argument `beta_prior_sd`). Finally, we change $\Sigma_{proposal}$ to have a value of 0.025, using the argument `proposal_sd`.

```r
#Generate object with parameters for covariate estimation
coeffs_obj  = mcleod.covariates.estimation.parameters(
                beta_init = 0,
                beta_prior_sd = 1.5,
                proposal_sd = 0.025)


comp_params = mcleod.computational.parameters(nr.gibbs = 3000)
# In addition to covariates, pass coeffs_obj as an argument
res = mcleod(x,
             n,
             covariates = covariates,
             covariates_estimation_parameters = coeffs_obj,
             computational_parameters = comp_params
             )
```

We obtain the posterior slope in the next code snippet. Note that we turn off the graphs for the posterior. Insead, we ask for a graph showing the MH proposals for $\vec{\beta}$. Accepted proposals are shown in red, while rejected proposals are shown in black.

```
# we retreive the posterior samples for the slopws.
# In addition, instead of plotting the posterior samples,
# we plot the PROPOSALS, with acccepted proposals shown
#in red, rejections in black
coeffs = mcleod::results.covariate.coefficients.posterior(res,
                plot.posterior = F,
                plot.MH.proposal.by.iteration = T)
```




```
print(coeffs$posterior.means)
```

```
## [1] -0.5054257
```

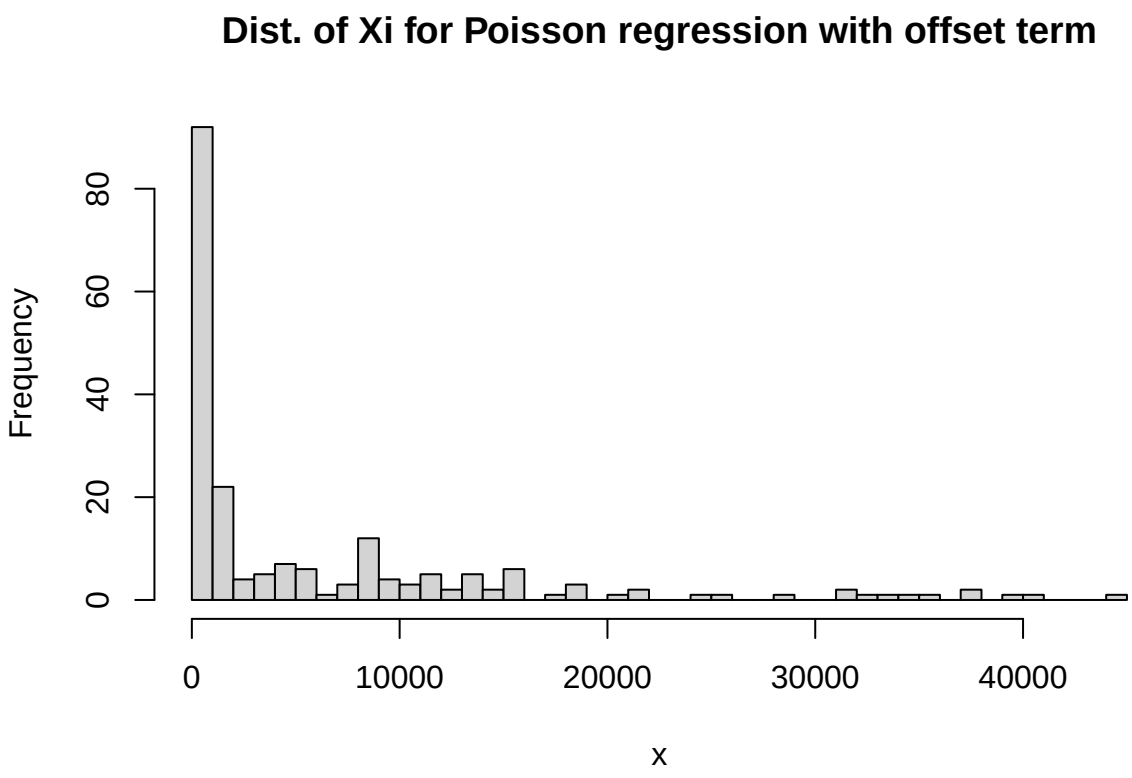
5.2 Poisson regression with a general random intercept and offset term

The next example shows how run a Poisson regression with a general random intercept term. The data is generated using a single covariate and a bi-modal distribution for the intercept. In addition, we include another extrinsic variable, C_i, so the model equation is of the form:

$$X_i \sim Pois(\lambda_i); log(\lambda_i) = \gamma_i + \beta Z_i + log(C_i).$$

The C_i's are known, and we show to incorporate them into the model using an offset term. The next code snippet generates and plots the data. Note the massive differences in orders of magnitude for X_i's, due to the C_i's, which are distributed uniformly between 1 and 100.

```
K = 200 #Number of samples
set.seed(2)
# A exponentially distributed covariate:
covariates = matrix(rexp(K,rate = 2),nrow = K)
# Value of the slope coefficient:
real_beta = 0.5

#indicator for the component in the bimodel intercept distribution.
u = sample(c(0,1),size = K,replace = T)
#draw a random intrinsic size, known to the user.
extrinsic_size = runif(n = K,1,100)
offset = log(extrinsic_size) #convert to log scale.

#generate data:
x = rpois(K,
    lambda = extrinsic_size * exp(rnorm(K,2 + 3*u,0.5) + real_beta* covariates)
          )

#Plot
hist(x,main='Dist. of Xi for Poisson regression with offset term',breaks = 50)
```



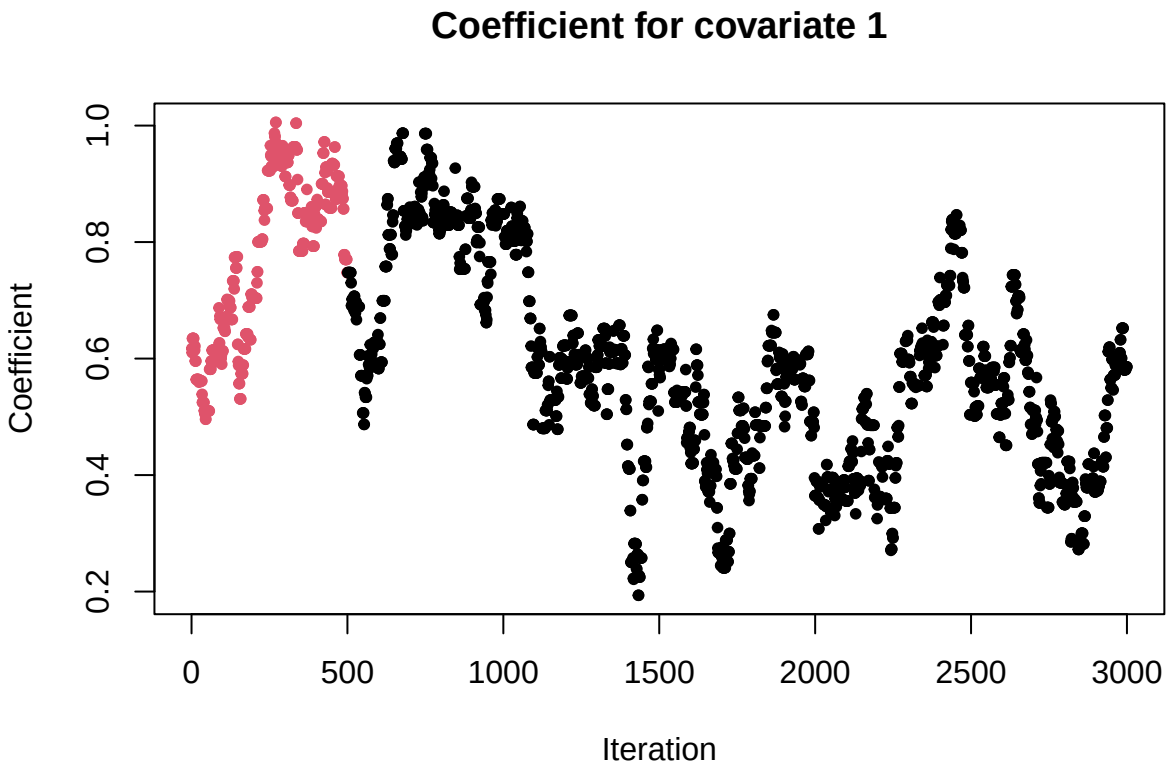
We run the Poisson regression by setting `Noise_Type = MCLEOD.POISSON.ERRORS`. The next code snippet also defines (a) the covariates, (b) the offset term and (c) the limits for $\vec{a}$.

```
#set the number of iterations
comp_obj = mcleod.computational.parameters(nr.gibbs = 3000,nr.gibbs.burnin = 500)
```

```
  res = mcleod(x, n.smp = NULL, #n.smp is null for Pois regression
               a.limits = c(-2,8), #set \vec{a}
               computational_parameters = comp_obj,
               covariates = covariates, # pass covariates
               Noise_Type = MCLEOD.POISSON.ERRORS, #Poisson regression
               offset_vec = offset #pass offset term
               )
```

We obtain the posterior distribution for the slope parameter, plot a graph of the MCMC samples, and print the posterior mean of the slope coefficient:

```
coeffs = mcleod::results.covariate.coefficients.posterior(res)
```

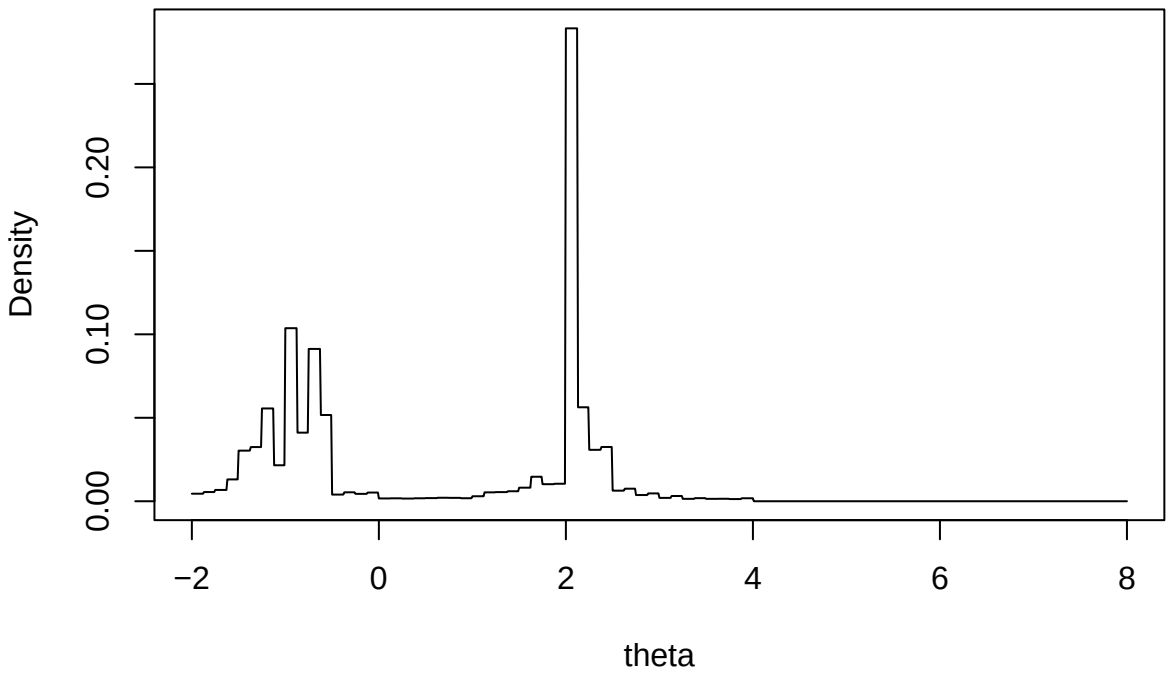



```
print(coeffs$posterior.means)
```

```
## [1] 0.577527
```

We also plot the estimate for the distribution of γ_i's:

```
posterior_dist_gamma = mcleod.get.posterior.mixing.dist(res)

x_points = seq(-2,8,0.01)
plot(x_points, posterior_mixing_dist$density(x_points),
     type = 'l',xlab = 'theta',ylab = 'Density')
```



5.2.1 Posterior estimates for γ_i

In the next code snippet, we obtain Posterior estimates for γ_i, for samples with $i = 1, 2, 3$ in the training data. The inputs include the observed X_i's, covariates, and offsets (values pass to argument `offset_vec` are $log(C_i)$). Note that we set `N` to `NULL`.

```
estimated.gamma_is = mcleod.posterior.estimates.random.effect(
  X = x[1:3],
  N = NULL,
  mcleod_res = res,
  covariates = covariates[1:3,,drop=F],
  offset_vec = offset[1:3])

#%% The estimated value for log(P_i/(1-P_i)):
estimated.gamma_is
```

```
## [1] 2.093753 5.771073 4.796210
```

5.2.2 Predictive intervals for X_i for a random intercept Poisson regression

Predictive intervals are obtained using the function `mcleod.predictive.interval`. The next code snippet computes predictive intervals for two observations: the first observation with $Z_1 = 0, C_1 = 1$; and the second observation with $Z_2 = -1, C_2 = 3$.

```
mcleod.predictive.interval(
  N = NULL,
  covariates = matrix(c(0, -1),ncol = 1),
```

```
  offset_vec = c(log(1) , log(3)),
  mcleod_res = res,
  Interval.Coverage = 0.95)
```

```
## $Lower
## [1] 2 3
##
## $Upper
## [1] 332.100 746.375
```

5.3 Setting the prior distributions on coefficient to be other than normal

In the final example for this section, we show how to set prior distribution for beta to other distribution than a multivariate normal distribution. The only constraint for the prior for $\vec{\beta}$ is that different components for the prior need to be independent.

We generate data with two covariates and a random normal intercept:

```
N = 30 #nr draws per sample
K = 200 #nr samples
set.seed(1)
covariates = matrix(rnorm(K*2,sd = 0.5),nrow = K) #generate covariates
real_beta_1 = -1 #true values for slopes
real_beta_2 = 1
#generate data:
x = rbinom(K,size = N,prob = inv.log.odds(rnorm(K,0,sd = 1)
          + real_beta_1*covariates[,1] + real_beta_2*covariates[,2]))
n = rep(N,K)
```

We use logistic regression to init the MCMC chain for $\vec{\beta}$:

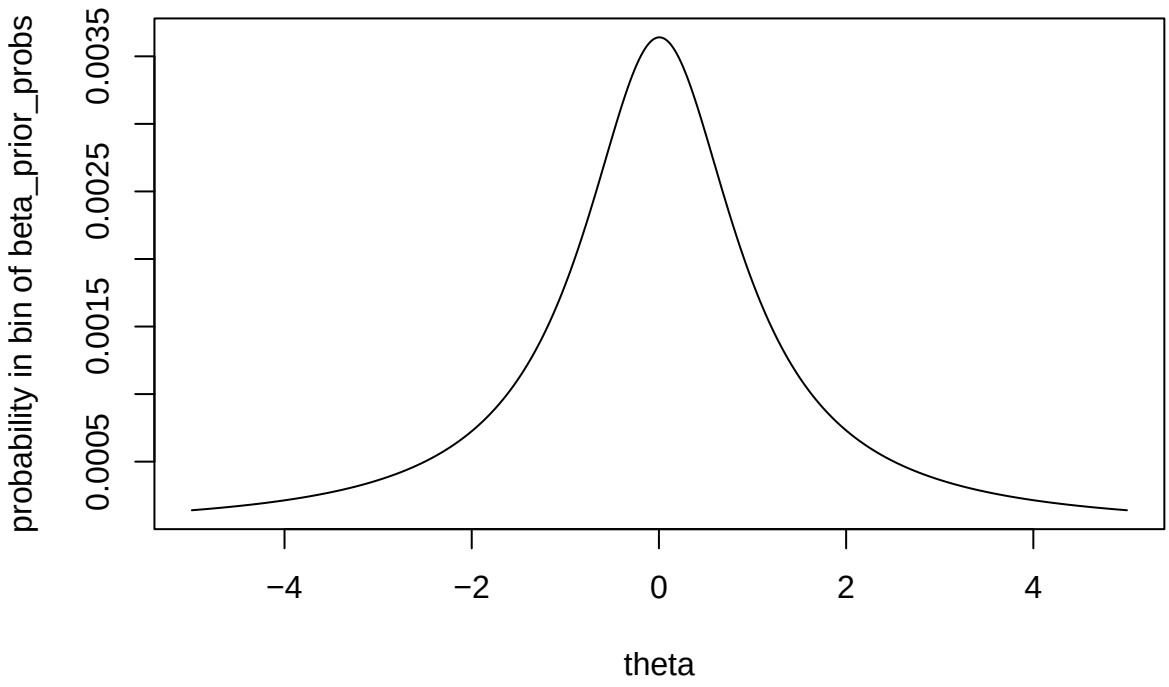
```
model_dt = data.frame(c = x,nc = n-x)
model_dt = cbind(model_dt,covariates)
model <- glm(cbind(c,nc) ~.,family=binomial,data=model_dt)
```

We define the prior for $\vec{\beta}$. Our code requires that the prior bu inserted as a series of partitioning points forming segments on the real line. In addition, the user must supply the probability for the slope parameter to be in any fo the segments. For this example, we assume a Cauchy prior for components of $\vec{\beta}$:

```
#we descretize the range (-5,5)
beta_prior_points = seq(-5,5,0.01)

# for each segment, we compute the probability to be in the segment
beta_prior_probs = pcauchy(beta_prior_points[-1]) -
                   pcauchy(beta_prior_points[-length(beta_prior_points)])

# we normalize, since we drop the tails out of (-5,5)
beta_prior_probs = beta_prior_probs/ sum(beta_prior_probs)
```

Next we plot the distribution. Our code will assume the prior is uniform within each segment:

```
 plot(beta_prior_points[-1],beta_prior_probs,
      type = 'l',xlab = 'theta',
      ylab = 'probability in bin of beta_prior_probs')
```



Since we have two components in $\vec{\beta}$, we pass a matrix for the probabilities. Rows correspond to segments formed between points in `beta_prior_points`, and columns correspond to components of $\vec{\beta}$. For this example we use the same prior for both slope coefficients, but this does not need to be the case for all times.

```
beta_prior_probs = matrix(c(beta_prior_probs, beta_prior_probs),ncol = 2)
```

We pass the support for the priors (`beta_prior_points`) and the density values in the different segments (`beta_prior_probs`) to `mcleod.covariates.estimation.parameters`:

```
coeffs_obj = mcleod.covariates.estimation.parameters(
      beta_init = model$coefficients[-1],
      Manual_Prior_Values = beta_prior_points,
      Manual_Prior_Probs = beta_prior_probs)
```

... and pass the resulting object as a parameter to `mcleod`:

```
res = mcleod(x, n, covariates = covariates,
               covariates_estimation_parameters = coeffs_obj)
```

Finally, we obtain the estimated slope parameters:

```
coeffs = mcleod::results.covariate.coefficients.posterior(res,plot.posterior = F)
coeffs$posterior.means
```

```
## [1] -0.8979254  0.8189644
```

6 Additional topics

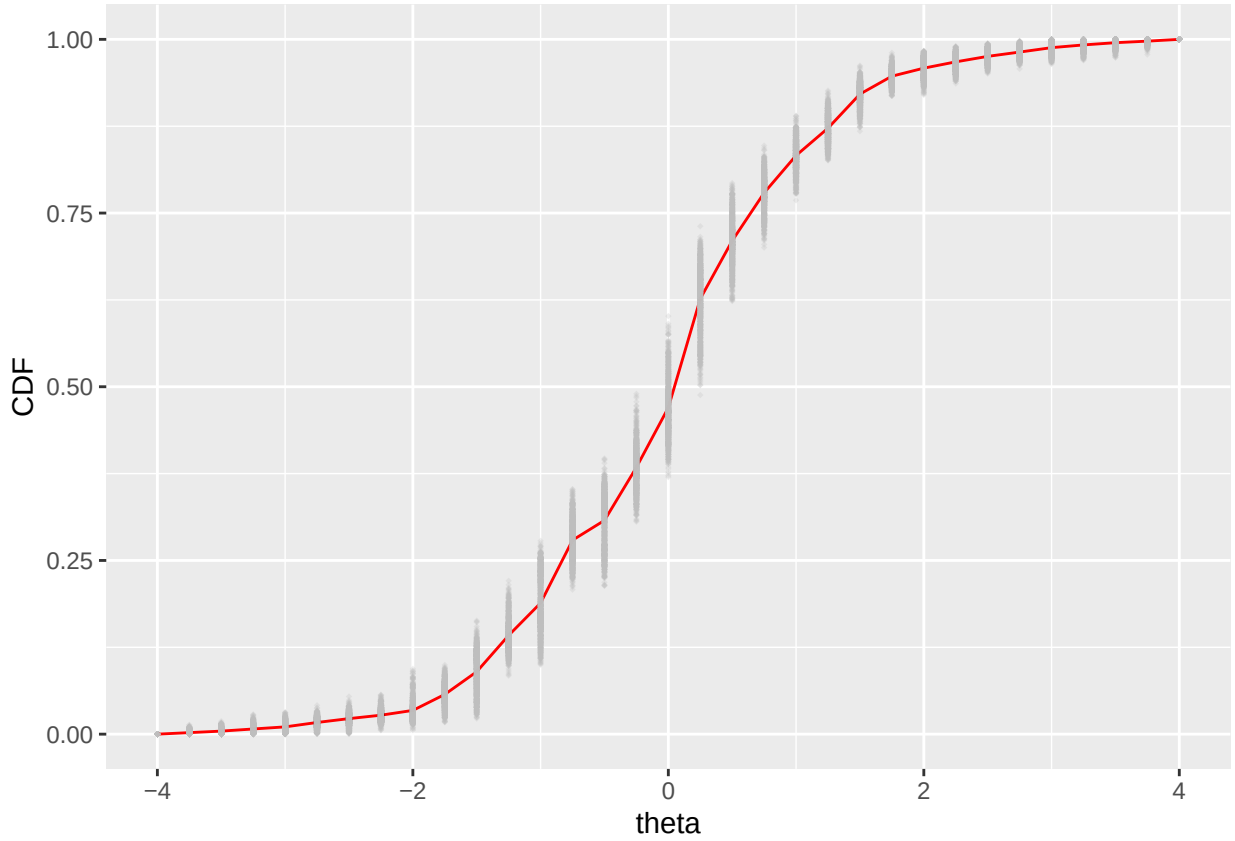
This section discusses additional advanced topics

6.1 Changing hyper-parameters for the priors of the mixing distribution:

We show to to change the hyper-parameters for the priors of the mixing distributions. We start with the following code snippet, which generates 2000 samples from a Beta-binomial distribution, with parameters $\alpha_1 = 2$, $\alpha_2 = 2$, and 20 draws per sample.

```
set.seed(1)

K = 2000 # Number of samples
n.vec = rep(20,K) #number of draws for each sample

# function for sampling a beta binomial (2,2)
generate_sample_Beta_2_2 = function(K){
  p.vec = rbeta(n = K,2,2)
  x.vec = rbinom(K,size = n.vec,p.vec)
  return(x.vec)
}

#Generate date
x.vec = generate_sample_Beta_2_2(K)

#run sampler:
mcleod_res = mcleod(x.vec,n.vec,a.limits = c(-4,4),
            # HBeta prior with 5 levels
            prior_param = mcleod.prior.parameters(
              prior.type = MCLEOD.PRIOR.TYPE.BETA.HEIRARCHICAL,
              Beta.Heirarchical.Levels = 5),
            # Set number of MCMC samples
            computational_parameters = mcleod.computational.parameters(
              nr.gibbs = 2000,nr.gibbs.burnin = 1000)
            )

mcleod::plot.posterior(mcleod_res)
```

```
## Warning: Removed 42 rows containing missing values (geom_point).
```

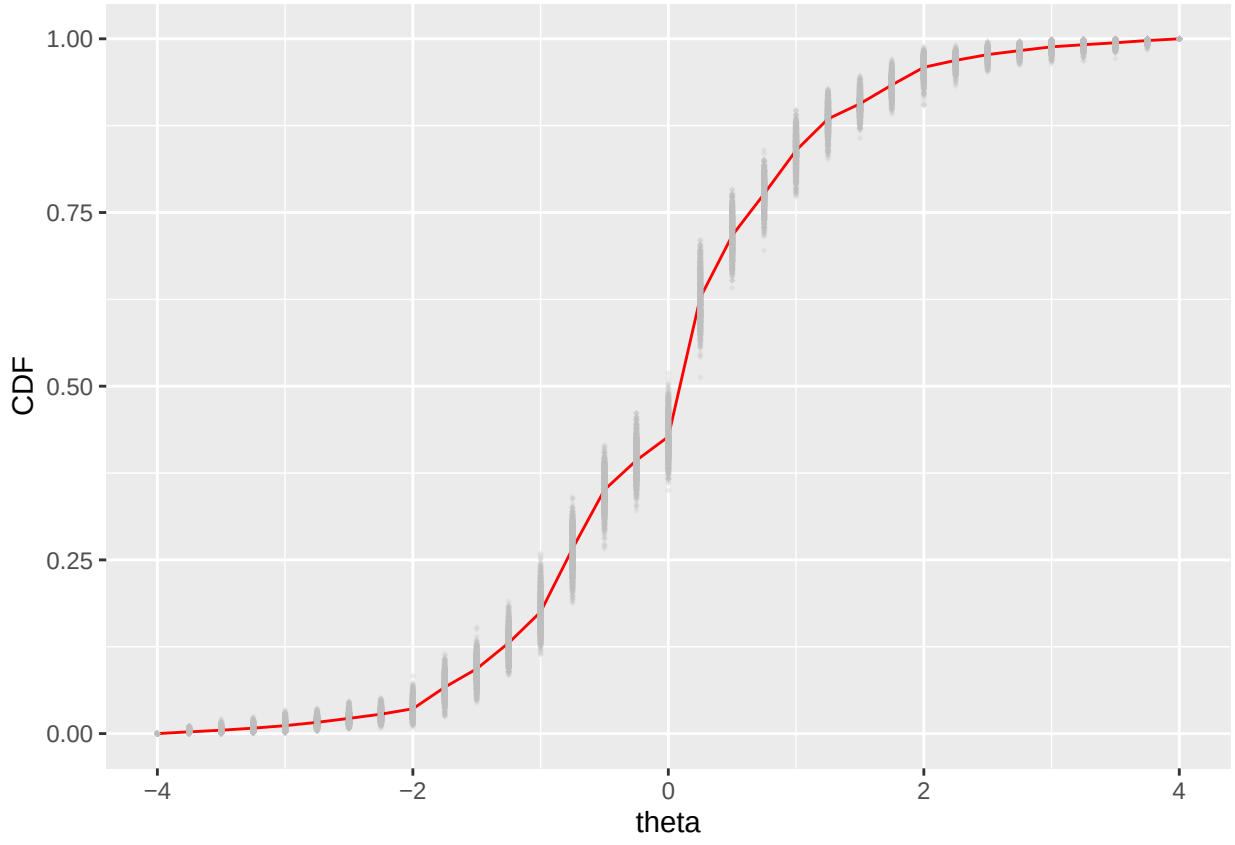


Next, we show how to change hyper parameters for the Polya tree (heirarchica Beta) prior. This involves passing two matrices of sizes $(L, 2^{(L-1)})$, one for the α_1 parameters and one for the α_2 parameters of the different nodes in the Polya tree. In each matrix, the lth row will define the corresponding parameters for the lth level of the tree. The $2^{(l-1)}$ leftmost parameters of the lth row will define the parameters of the nodes on the lth level, from left to right.

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#Example how to change hyper-parameters.
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#For, example, how to change all hyper parameters to 2
# We form two matrices, for $\alpha^{L}$ and $\alpha^{U}$.
# In each matrix, the [l,i] entry gives the appropriate hyper parameter
# for the ith node on the lth level.

L = 5 # number of levels
alpha_L = matrix(NA,nrow = L,ncol = 2^(L-1)) # hyper-parameter matrices
alpha_U = matrix(NA,nrow = L,ncol = 2^(L-1))

# Fill all entries to a value of 2
for(l in 1:L){
  for(k in 1:2^(l-1)){
    alpha_L[l,k] = 2
    alpha_U[l,k] = 2
  }
}

# we pass the hyper-paramers to the mcleod.prior.parameters(...) function
```

```
mcleod_res = mcleod(x.vec,n.vec,
    a.limits = c(-4,4),
    #pass prior hyper parameters
    prior_param = mcleod.prior.parameters(
          prior.type = MCLEOD.PRIOR.TYPE.BETA.HEIRARCHICAL,
          Beta.Heirarchical.Levels = L,
          #NOTE: THESE ARE THE MATRICES FOR THE HYPER-PARAMETERS:
          Prior_Hyper_Parameters_BetaH_L = alpha_L,
          Prior_Hyper_Parameters_BetaH_U = alpha_U),
    # set the number of iterations
    computational_parameters = mcleod.computational.parameters(
      nr.gibbs = 2000,
      nr.gibbs.burnin = 1000)
  )

mcleod::plot.posterior(mcleod_res)
```

```
## Warning: Removed 44 rows containing missing values (geom_point).
```

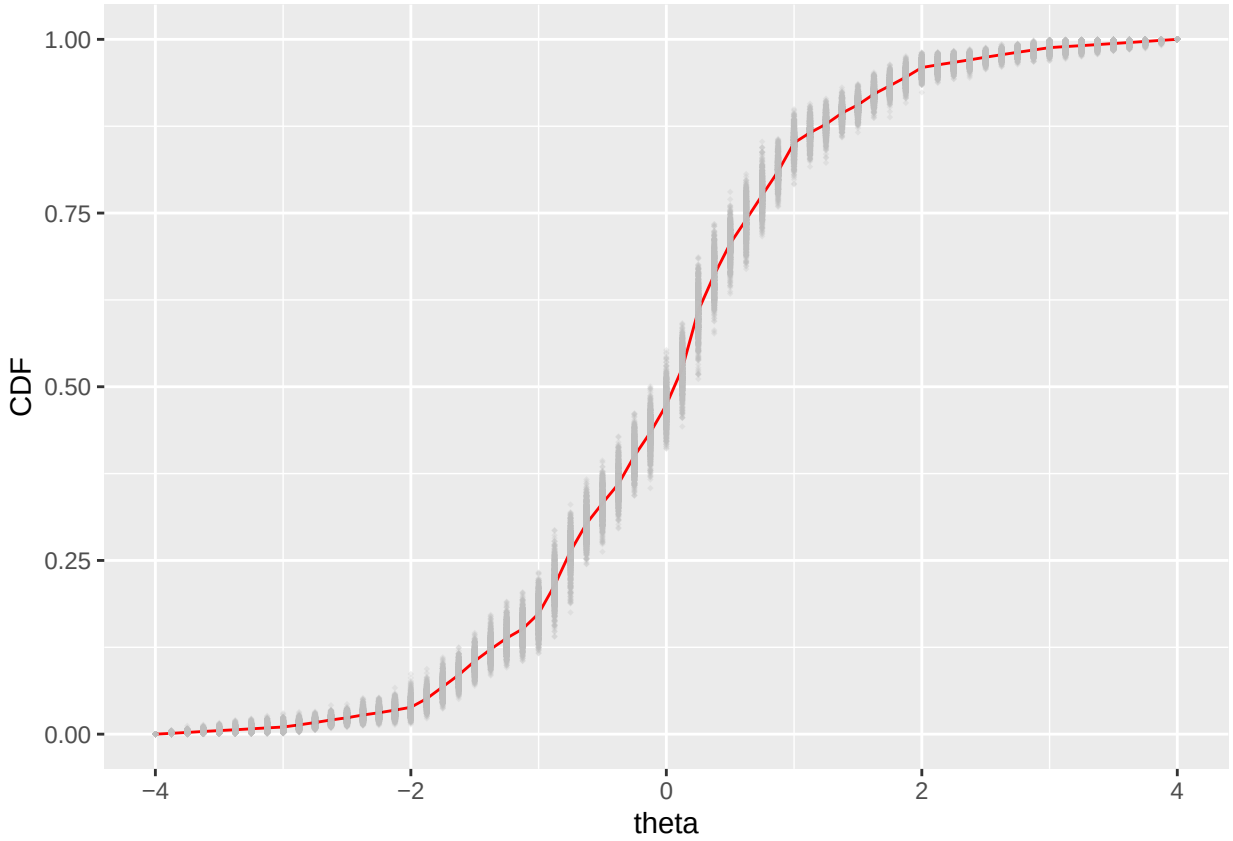


We got a slightly different mixing distribution for the same data.

Next, we show how the change the hyper parameters for the two-level Dirichlet Tree.This requires passing a single matrix with two rows and number of columns corresponding to the number of segments in the mixing distribution. The first row sets the hyper-parameters for the Dirichlet variable at the top of the tree. The second row sets the hyper-parameter for the Dirichlet variables at the middle level. Values correspond to segments from left to right. Each `Two.Layer.Dirichlet.Intervals / Two.Layer.Dirichlet.Nodes.in.First.Layer` set of values is used to form a Dirichlet variable.

```
# Set tree structure
Nodes_in_first_layer = 8
Total_nr_nodes = 64

#set matrix for hyper parameters.
Two_Level_Dirichlet_Tree_Hyperparameters = matrix(NA,nrow = 2,ncol = Total_nr_nodes)

#Dirichlet at top of tree is Dirichlet(2,2,2,2,2,2,2,2)
Two_Level_Dirichlet_Tree_Hyperparameters[1,1:Nodes_in_first_layer] = 2

# At the middle level we have 8 Dirichlet random variables.
# We set all of them to be Dirichlet(2,2,2,2,2,2,2,2) as well.

Two_Level_Dirichlet_Tree_Hyperparameters[2,1:Total_nr_nodes] = 2

# we pass the hyper-paramers to the mcleod.prior.parameters(...) function
mcleod_res = mcleod(x.vec,n.vec,
    a.limits = c(-4,4),
    exact.numeric.integration = T,
    #Define prior
    prior_param = mcleod.prior.parameters(
      #Define as 2-layer Dirichlet tree
      prior.type = MCLEOD.PRIOR.TYPE.TWO.LAYER.DIRICHLET,
      #set tree structure
      Two.Layer.Dirichlet.Intervals = Total_nr_nodes,
      Two.Layer.Dirichlet.Nodes.in.First.Layer =  Nodes_in_first_layer,
      #NOTE: HERE WE PASS THE HYPER-PARAMETERS
      Prior_Hyper_Parameters_2LDT = Two_Level_Dirichlet_Tree_Hyperparameters),

    #Set the number of iterations
    computational_parameters = mcleod.computational.parameters(
      nr.gibbs = 2000,
      nr.gibbs.burnin = 1000)
)

mcleod::plot.posterior(mcleod_res)
```

```
## Warning: Removed 122 rows containing missing values (geom_point).
```


1.00
0.75
0.50
0.25
0.00
CDF
−4
−2
0
2
4
theta