Building a Gibbs Sampler with dbarts

Vincent Dorie

12/21/2020

For users who are comfortable with writing their own posterior samplers, `dbarts` makes it easy to incorporate a linear BART component in a Gibbs sampler. In the example below, we fit a functional mixture model, where observations may have come from one of two underlying functions and class membership is unobserved.

Model

Performing inference conditional on x, we assume that:

$$Y \mid Z = z \sim \mathrm{N}\left(f(x, z), \sigma^2\right),$$
$$Z \sim \mathrm{Bern}(p),$$

and that Z is unobserved. To make the model fully Bayesian, we impose a $\mathrm{Beta}(1, 1)$ prior on p. f and σ^2 have implicit priors defined by BART.

Conditonal Posteriors

To implement a Gibbs sampler, we will have to produce updates for Z and p. After writing down the joint distribution of Z, Y, p, f, and σ^2, we have:

- $P(Z = 1 \mid Y = y, p, f, \sigma^2) \propto \phi(y, f(x, 1), \sigma) \cdot p$
- $P(Z = 0 \mid Y = y, p, f, \sigma^2) \propto \phi(y, f(x, 0), \sigma) \cdot (1 - p)$
- $p \mid Z = z \sim \mathrm{Beta}(1 + \sum z_i, 1 + \sum(1 - z_i))$

where ϕ is the density of a normal distribution. This yields an overall strategy of:

1. Draw a sample from BART
    1. Draw samples of $f(x, 1)$ and $f(x, 0)$
    2. Draw a sample of σ^2
2. Use $f(x, 1)$, $f(x, 0)$, and p to draw a sample of Z
3. Use Z to draw a sample of p

Simulated Data

To illustrate, we create some toy data. The simulated data is visualized later, together with the fitted model.

```
# Underlying functions
f0 <- function(x) 90 + exp(0.06 * x)
f1 <- function(x) 72 + 3 * sqrt(x)

set.seed(2793)

# Generate true values.
n <- 120
p.0 <- 0.5
```

```
z.0 <- rbinom(n, 1, p.0)
n1 <- sum(z.0); n0 <- n - n1

# In order to make the problem more interesting, x is confounded with both
# y and z.
x <- numeric(n)
x[z.0 == 0] <- rnorm(n0, 20, 10)
x[z.0 == 1] <- rnorm(n1, 40, 10)
y <- numeric(n)
y[z.0 == 0] <- f0(x[z.0 == 0])
y[z.0 == 1] <- f1(x[z.0 == 1])
y <- y + rnorm(n)

data_train <- data.frame(y = y, x = x, z =     z.0)
data_test  <- data.frame(       x = x, z = 1 - z.0)
```

Implementing a Gibbs Sampler

For this specific model, the sampler needs to be updated with new predictor variables/covariates. This can be done with a `dbartsSampler` reference class object by calling the `sampler$setPredictor` function. As, given any $Z = z$, the sampler also needs to evaluate $f(x, 1 - z)$, we utilize the test slot of the sampler and keep it up-to-date with the "counterfactual" predictor using the `sampler$setTestPredictor` function. Models that instead modify the response variable can use the `sampler$setResponse` or `sampler$setOffset` functions.

One complication in updating the predictor matrix while the sampler runs is that new values of z must leave the sampler in an internally consistent state. During warmup this constraint is ignored, however, after warmup is complete rejection sampling is used to guarantee that no leaf nodes are empty. This is accomplished by using the `forceUpdate` argument to `setPredictor` and checking that the logical response is `TRUE`.

```
n_warmup  <- 100
n_samples <- 500
n_total   <- n_warmup + n_samples

# Allocate storage for result.
samples_p   <- rep.int(NA_real_, n_samples)
samples_z   <- matrix(NA_real_, n_samples, n)
samples_mu0 <- matrix(NA_real_, n_samples, n)
samples_mu1 <- matrix(NA_real_, n_samples, n)

library(dbarts, quietly = TRUE)

# We only need to draw one sample at a time, although for illustrative purposes
# a small degree of thinning is done to the BART component.
control <- dbartsControl(updateState = FALSE, verbose = FALSE,
                         n.burn = 0L, n.samples = 1L, n.thin = 3L,
                         n.chains = 1L)
# We create the sampler with a z vector that contains at least one 1 and one 0,
# so that all of the cut points are set correctly.
sampler <- dbarts(y ~ x + z, data_train, data_test, control = control)

# Sample from prior.
p <- rbeta(1, 1, 1)
z <- rbinom(n, 1, p)
```

```
# Ignore result of this sampler call
invisible(sampler$setPredictor(x = z, column = 2, forceUpdate = TRUE))
sampler$setTestPredictor(x = 1 - z, column = 2)
sampler$sampleTreesFromPrior()

for (i in seq_len(n_total)) {
  # Draw a single sample from the posterior of f and sigma^2.
  samples <- sampler$run()

  # Recover f(x, 1) and f(x, 0).
  mu0 <- ifelse(z == 0, samples$train[,1], samples$test[,1])
  mu1 <- ifelse(z == 1, samples$train[,1], samples$test[,1])

  p0 <- dnorm(y, mu0, samples$sigma[1]) * (1 - p)
  p1 <- dnorm(y, mu1, samples$sigma[1]) * p
  p.z <- p1 / (p0 + p1)
  z <- rbinom(n, 1, p.z)

  if (i <= n_warmup) {
    sampler$setPredictor(x = z, column = 2, forceUpdate = TRUE)
  } else while (sampler$setPredictor(x = z, column = 2) == FALSE) {
    z <- rbinom(n, 1, p.z)
  }
  sampler$setTestPredictor(x = 1 - z, column = 2)

  n1 <- sum(z); n0 <- n - n1
  p <- rbeta(1, 1 + n0, 1 + n1)

  # Store samples if no longer warming up.
  if (i > n_warmup) {
    offset <- i - n_warmup
    samples_p[offset] <- p
    samples_z[offset,] <- z
    samples_mu0[offset,] <- mu0
    samples_mu1[offset,] <- mu1
  }
}
```

Saveing the sampler

If it desired to use `save` and `load` on the sampler, it is required to instruct sampler stored using the reference class to write its state out as an `R` object:

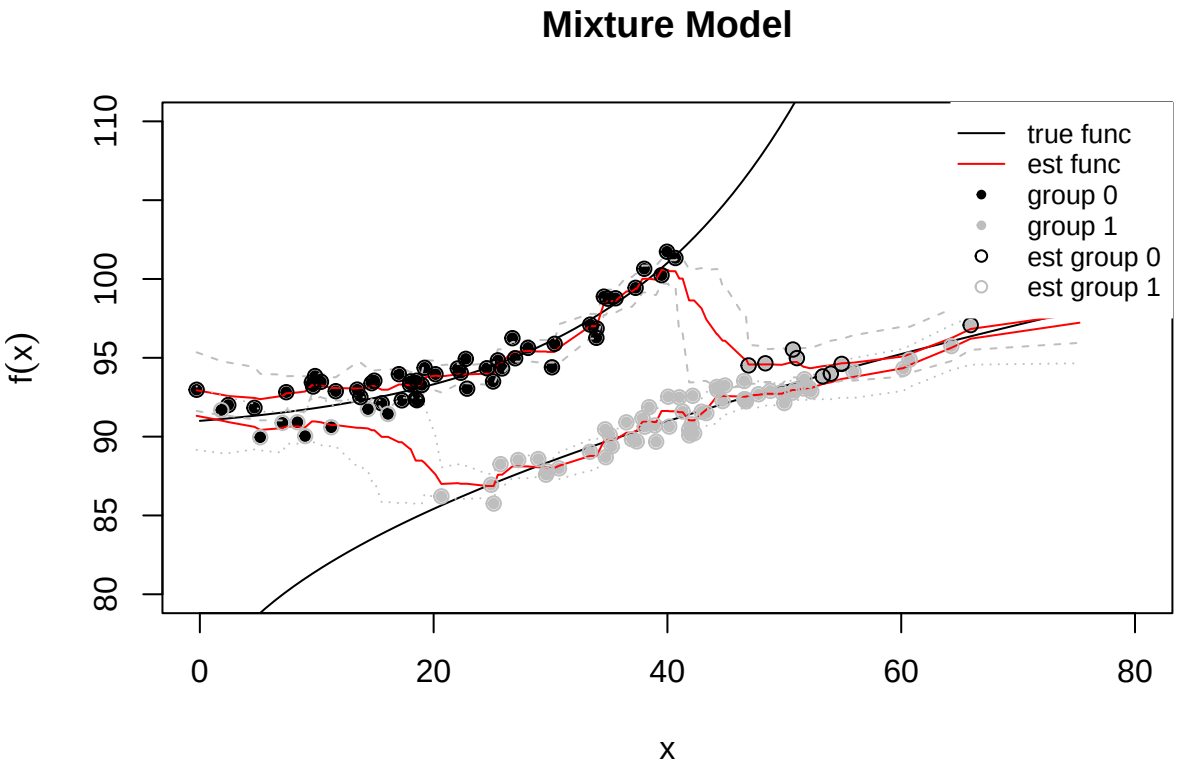
```
sampler$storeState()
```

Visualizing the Result

Finally, we visualize the results. In the graph below, the estimated label of observations encircles the true labels, both of which are represented by point color. We see that, beyond the range of the observed data, the estimated functions regress towards each other and points start to be mislabeled.

```
mean_mu0 <- apply(samples_mu0, 2, mean)
mean_mu1 <- apply(samples_mu1, 2, mean)
```

```r
ub_mu0 <- apply(samples_mu0, 2, quantile, 0.975)
lb_mu0 <- apply(samples_mu0, 2, quantile, 0.025)

ub_mu1 <- apply(samples_mu1, 2, quantile, 0.975)
lb_mu1 <- apply(samples_mu1, 2, quantile, 0.025)

curve(f0(x), 0, 80, ylim = c(80, 110), ylab = expression(f(x)),
      main = "Mixture Model")
curve(f1(x), add = TRUE)

points(x, y, pch = 20, col = ifelse(z.0 == 0, "black", "gray"))

lines(sort(x), mean_mu0[order(x)], col = "red")
lines(sort(x), mean_mu1[order(x)], col = "red")

# Add point-wise confidence intervals.
lines(sort(x), ub_mu0[order(x)], col = "gray", lty = 2)
lines(sort(x), lb_mu0[order(x)], col = "gray", lty = 2)

lines(sort(x), ub_mu1[order(x)], col = "gray", lty = 3)
lines(sort(x), lb_mu1[order(x)], col = "gray", lty = 3)

# Without constraining z for an observation to 0/1, its interpretation may be
# flipped from that which generated the data.
mean_z <- ifelse(apply(samples_z, 2, mean) <= 0.5, 0L, 1L)
if (mean(mean_z != z.0) > 0.5) mean_z <- 1 - mean_z

points(x, y, pch = 1, col = ifelse(mean_z == 0, "black", "gray"))
legend("topright", c("true func", "est func", "group 0", "group 1",
                     "est group 0", "est group 1"),
       lty = c(1, 1, NA, NA, NA, NA), pch = c(NA, NA, 20, 20, 1, 1),
       col = c("black", "red", "black", "gray", "black", "gray"),
       cex = 0.8, box.col = "white", bg = "white")
```



Multiple Threading

When implementing a Gibbs sampler using `dbarts` in R, it is most often the case that multiple threads will need to handled by creating separate copies of the sampler and data. While `dbartsSampler`s are natively multithreaded, few of their slots are stored independently across chains. For an example of this approach and a more complete implementation of the principles in this document, consult the implementation of rbart_vi.