Bayesian Additive Tree Regression

Possible extension for treatment effect estimation

Introduction to Bayesian Trees

Classification and regressions trees are widely used supervised learning methods. Their popularity stems from model interpretability and prediction performance. Tree methods partition predictor space into a disjoint set of regions, whose union describes the full predictor space. The partition takes place through splitting rules on X and effectively distributes X to the corresponding region. Each region is then fitted with a model, such as the response variable mean for the Xs that populate each region. The conditional distribution of the response variable Y given predictor variables X is specified by the tree's splitting rules subject to a chosen fitting criteria, such as minimizing region mean squared error. Regression trees relate predictors to a quantitative response while classification trees relate predictors to a categorical or binary response. This analysis focuses on regression trees.

Terminal nodes, or the partitioned regions, are used for regression tree prediction. Let $y = f(x) + \varepsilon$ be the model describing the relationship between X and Y. Notice that we make no assumption on model functional form aside from having additive errors ε that are distributed $\mathcal{N}(0, \sigma^2)$. $f(x)$ is an unknown function that we want to "learn" from data. Classical regression tree methods take a nonparametric approach to approximate $f(x) = E(y|x)$ through splitting rules. The prediction of y from x is then

$$\hat{f}(x) = \sum_{b=1}^{B} \hat{c}_b I(x \in R_b)$$

where B denotes the number of terminal nodes, $\hat{c}_b$ is the node estimator, and $I(x \in R_b)$ is the indicator function for x being in region B. The node estimator $\hat{c}_b$ of c_b is mean$(y|x \in R_b)$. The collection of node estimates is described by $M = \{\mu_1, ..., \mu_B\}$. $f(x)$ describes a single tree T with estimators M. Using different notation for $f(x)$ in order to appropriately describe it based on T and M, we let $f(x) \Leftrightarrow g(x; T, M)$. We can then describe the set of many trees $\mathcal{G} = \{g(x; T_1, M_1), ..., g(x; T_V, M_V)\}$ based on each tree's nodes $M_v = \{\mu_{v1}, ..., \mu_{vB}\}$ for $v = 1, ..., V$.

Moving towards generalized additive models $E(Y|x_1, ..., x_p) = \sum_{j=1}^{p} f_j(x_j)$, the sum-of-trees model becomes $E(Y|X) = \sum_{v=1}^{V} g_v(X; T_v, M_v)$. Ensemble methods are currently used to evaluate a set of trees. These methods include bagging, which stochastically generates independent trees to reduce prediction variation by averaging, and boosting, which fits a linear combination of trees for the purpose of explaining variation not captured by the previous fit in order to reduce bias. Random forest methods are also used to evaluate a set of trees and are included in the bagging category. Bayesian tree model averaging produces a semiparametric approach for evaluating a set of trees. Averaging is conducted over a distribution of terminal node posterior samples, generated from a Bayesian tree environment. Bayesian additive regression tree (BART) methods are a direct extension of the single-tree Bayesian tree model (Chipman et al. 2010).

BART has received recent attention for its causal inference ability when evaluating treatment and control outcomes (Hill 2011). In these settings a researcher uses treatment and control samples to construct estimators for quantities of interest like average treatment effects. However, counterfactual problems arise when trying to compute actual treatment effects due to a latent variable structure on observations. The objective of this project (possible paper?) is to extend BART sampling methods by allowing for latent variable predictor space structure. To (possibly) accomplish this objective, I propose a sampling technique that extends BART's existing MH-Gibbs sampler to a EM-MH-Gibbs sampler.

Incorporating an EM algorithm into the BART sampling structure adjusts terminal node posterior form in order to capture uncertainty surrounding unobservable effects. While having full information on treatment and control outcomes negates the need to model uncertainty, in observational settings researchers lack this information. While IV estimation techniques have been widely used, satisfying exclusion restrictions remains subjective. Developing the BART EM-MH-Gibbs estimator contributes to the econometric causal inference field, and statistics field more generally, by producing a technique that lets treatment effect information reveal itself from original data. Specifying informative prior distributions for parameters in Bayesian settings

is also a subjective endeavor, but allowing stochastic hyperparameters reduces the effect of prior specification, and this is the approach taken for developing an EM addition to existing BART methods.

A goal is for this model to be used for causal inference in policy settings. For example, policy makers can use historical data to see effects of previous policies and construct future policies that have targeted or more efficient outcomes. Policies seeking nutrient availability interventions in developing areas or livestock health interventions in production systems immediately come to mind. However, the flexibility of this estimator does not constrain its use to any particular research field.

The first step is replicating a BART model, which then allows for estimator extension. A BART model is characterized by a sum-of-trees and approximates $E(Y|X)$ through

$$f(x) = g(x; T_1, M_1) + g(x; T_2, M_2) + ... + g(x; T_V, M_V).$$

A prior on T and M is specified in order to minimize individual tree effects on the full conditional $Y|X$. The result is that each tree accounts for small portions of variation in Y. Each portion of variation is modeled through Bayesian backfitting a Markov chain Monte Carlo sample (Hastie and Tibshirani 2000) by iteratively constructing residuals on the ith iteration and using these residuals to fit the next $i+1$ iteration. The number of trees V generated for the MCMC sample is user chosen. Independence across priors is assumed, providing

$$\begin{aligned} p((T_1, M_1), ..., (T_V, M_V), \sigma) &= \left(\prod_v p(T_v, M_v) \right) p(\sigma) \\ &= \left(\prod_v p(M_v|T_v) p(T_v) \right) p(\sigma) \\ &= \left(\prod_v \prod_b p(\mu_{bv}|T_v) p(T_v) \right) p(\sigma). \end{aligned}$$

The prior $p(T)$ on T is composed of a terminal node splitting part p_{split} that characterizes the probability of creating a new node from an existing terminal node, and a splitting rule part p_{rule} that draws a random splitting rule informed by X. The probability a terminal node splits is

$$p_{\text{split}}(\eta, T) = \alpha(1 + d_\eta)^{-\beta},$$

where d represents the depth of node η for tree T, and $\alpha \in (0, 1)$, $\beta \geq 0$ are constants acting as tuning parameters. Growing a tree is a decreasing function of current node depth, which keeps trees from growing too large. Chipman, George, and McCulloch (1998) (CGM98) tested a grid of α and β values and found $\{\alpha, \beta\} = \{0.95, 2\}$ to be effective in keeping individual tree sizes relatively small when incorporated in a sum-of-trees model. The splitting rule ρ for determining $p_{\text{rule}}(\rho|\eta, T)$ is described by the distribution of available predictors, and conditional on the predictors, the available splits defined by the range of each predictor. A splitting rule is then defined as a uniform draw across available predictors followed by a uniform draw of possible split values from the range of the drawn predictor. Splitting rules cannot be constructed in a way that leaves terminal nodes empty. For example, if a binary variable has been previously drawn and used in the tree, it cannot be drawn again.

The prior $p(\mu_{vb}|T_v)$ for M is $\mathcal{N}(0, \sigma_\mu^2)$, where $\sigma_\mu^2 = 0.5/(k\sqrt{V})$. CGM98 recommend $k = 2$ as a default so that there is an approximate 95% prior probability of $E(Y|x)$ being within the interval $(Y_{\min}, Y_{\max})$. CGM98 rescale the response data Y to be in the interval $(-0.5, .5)$ to support ease of prior form and computation. However, rescaling Y is not necessary. $p(\mu_{vb}|T_v)$ would still be chosen to follow $\mathcal{N}(\cdot)$ but $\mu_\mu = 0$ would be replaced with $\mu_\mu = \theta$, which would then either be defined as a constant such as $\bar{Y}$, or θ would be allowed to vary and follow a specified hyperparameter prior distribution. For the purpose of this part of the analysis, priors are chosen to follow CGM98 defaults.

The prior $p(\sigma)$ on σ is chosen as $\nu\lambda(\text{inv} - \chi^2(\nu))$, which is equivalently inv-gamma$(\nu/2, \nu\lambda/2)$. The shape ν is chosen between values 3 to 10, with 3 being the recommended default. λ is informed by the response

variable through $\hat{\sigma}$, which is either estimated as the standard deviation of Y or as the estimated standard deviation of residuals from a linear regression of Y on X. λ is then chosen such that $P(\sigma < \hat{\sigma}) = q$ for a specified quantile q typically set at 0.9.

The likelihood on Y is specified as $y_{b1}, ..., y_{bn} \overset{\text{iid}}{\sim} \mathcal{N}(\mu_b, \sigma^2)$ for $b = 1, ..., B$. Combining all previous information results in

$$p(T)p(Y|X,T) \approx p(T) \prod_{b=1}^{B} \frac{(\nu\lambda)^{\frac{\nu}{2}}}{\pi^{\frac{n_i}{2}}} \sqrt{\frac{a}{n_i + a}} \frac{\Gamma\left(\frac{n_i+\nu}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} (s_i + t_i + \nu\lambda)^{-\frac{n_i+\nu}{2}}$$

with $s_i = (n_i - 1)\text{var}(Y_i)$, $t_i = ((n_i a)/(n_i + a))(\bar{y}_i^2)$, and $a = 1/(k\sqrt{m})$.

Generating the posterior chain $\{(T_v, M_v)\}_{v=1}^{V}$ is accomplished by using a MH-Gibbs step that first accepts a proposal tree T^* for the $T_{(v+1)}$ iteration or keeps the previously generated tree $T_{(v)}$ with probability ϕ, and then updates with $M_v = \{\mu_{v1}, ..., \mu_{vB}\}$ draws from $p(\mu_{vb}|T_v)$ depending on the size B of terminal nodes for the accepted tree T^* or $T_{(v)}$ for the current iteration. The proposal densities, or transition kernels, for generating a tree T^* from T, $q(T, T^*)$, and generating a tree T from T^*, $q(T^*, T)$, are described by the different transition possibilities:

grow - randomly pick a terminal node and assign a splitting rule using $p_{\text{rule}}(\rho|\eta, T)$

prune - randomly pick an internal parent of terminal nodes and collapse the terminal nodes so that the parent is the new terminal node

change - randomly pick an internal parent node and reassign it a splitting rule using $p_{\text{rule}}(\rho|\eta, T)$

swap - randomly pick an internal parent node and swap its splitting rule with the splitting rule of the child nodes

After acceptance of the proposal tree or previous tree and the corresponding draws of M_v, the residuals are computed for that iteration and are used as the response data in the next iteration. Formally, the MH-Gibbs algorithm is

Initialize residuals $R_0 = Y$

(i) Generate T^* from $q(T_{(v)}, T^*)$

(ii) Take $T_{(v+1)} = T^*$ with probability $\delta(T_{(v)}, T^*)$[1], else $T_{(v+1)} = T_{(v)}$

where $\delta(T_{(v)}, T^*) = \min\left\{\frac{q(T^*, T_{(v)})}{q(T_{(v)}, T^*)} \frac{p(R_v|X, T^*)}{p(R_v|X, T_{(v)})} \frac{p(T^*)}{p(T_{(v)})}, 1\right\}$

(iii) Draw M_v per $B \in T^*$ or $T_{(v)}$

(iv) Compute $R_{v+1} = Y - \sum_{l \neq v+1} g(X; T_l, M_l)$

(v) After a chain of size V has been generated, draw σ

BART replication proceeds by following Hill (2010), who generates BART estimates for generated data using the `BayesTree` R package developed by Chipman, George, and McCulloch. For variables Y, Z, and X, the data generating process follows:

$X|Z = 1 \sim \mathcal{N}(40, 10^2)$

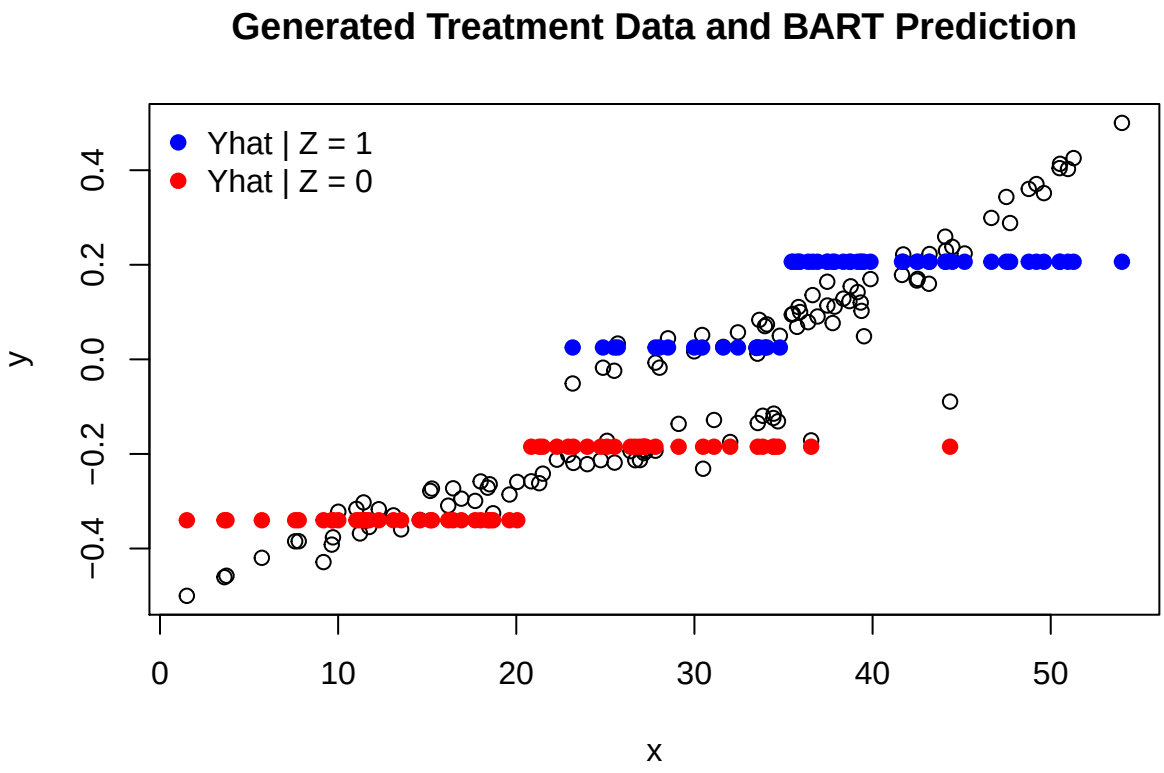
$Y_{(Z=1)}|X \sim \mathcal{N}(90 + e^{0.06X}, 1)$

$X|Z = 0 \sim \mathcal{N}(20, 10^2)$

$Y_{(Z=0)}|X \sim \mathcal{N}(72 + 3\sqrt{X}, 1)$

[1] This is accomplished by a Bernoulli draw with probability δ where acceptance occurs if the draw is 1.

Following the above algorithm in CGM98, BART estimates are developed from scratch in R[2]. The plot below provides fitted estimates from one BART chain averaged over posterior draws and successfully replicates the plot in Figure 2 on page 6 of Hill (2010) showing single-tree BART estimates[3].



EM Extension to MCMC Bayesian Tree Posterior Sampling

The EM extension to CGM98's MH-Gibbs algorithm removes the conditioning on Hill's (2010) treatment variable $Z = \{0, 1\}$. This approach specifies a latent structure on Y. The original specification $y = f(x, z) + \varepsilon$ is respecified as $y = f(x, \omega) + \varepsilon$ where ω represents a $\mathcal{N}(\theta_\mu, \sigma_\mu^2)$ unobservable effect of Z on Y through X. The latent variable ω is truncated at some value c. In order to not introduce bias by directly choosing the truncation point, c is also specified as a parameter with an appropriate prior distribution $p(c)$. This creates an EM step in the existing MH-Gibbs algorithm. The likelihood of Y is augmented to reflect a random ω, $p(Y|X, T, \omega)p(\omega)p(c)$. The "complete" likelihood on Y can be computed using additional information from an EM procedure that iteratively maximizes the expectation of latent variable structure Y with respect to θ_μ. This is incorporated in the MH step of generating trees, whose first split variable assignment depends on ω. It is the occurance of this first splitting variable in the generated chain of trees that allows evaluation of treatment effect estimators.

With Y following a normal distribution, censoring Y at some value c results in the unobservable variable Z following a truncated normal disribution. Maximizing the complete likelihood in Y with respect to θ_μ is completed by iteratively altering the FOC expresion

$$\hat{\theta} = \frac{m\bar{y} + (n - m)E(Z)}{n},$$

where n represents the full sample size and m represents the truncated portion of the sample size. The expecation of Z is with respect to the conditional density $Z|\theta, X$ and is characterized by the truncated

[2]Code is appended to the end of this document.
[3]Replicated up to random perturbations from the data generation process.

normal density $\phi(c-\theta)/(1-\Phi(c-\theta))$, where ϕ and Φ represent normal disribution and density functions, respectively. The full characerization becomes

$$\hat{\theta}^{(j+1)} = \frac{m}{n}\bar{y} + \frac{(n-m)}{n}\left(\hat{\theta}^{(j)} + \frac{\phi(c-\hat{\theta}^{(j)})}{1-\phi(c-\hat{\theta}^{(j)})}\right)$$

and continues in an iterative process until a specified tolerance is met.

The EM implementation in the MH algorith proceeds as follows.

(i) Generate a sample Y_s of size n with replacement and draw $c \sim \mathcal{U}(\min(Y_s), \max(Y_s))$

(ii) Given $\{\bar{y}, c, \theta\} = \{\bar{y}_s, c_s, \theta_0\}$

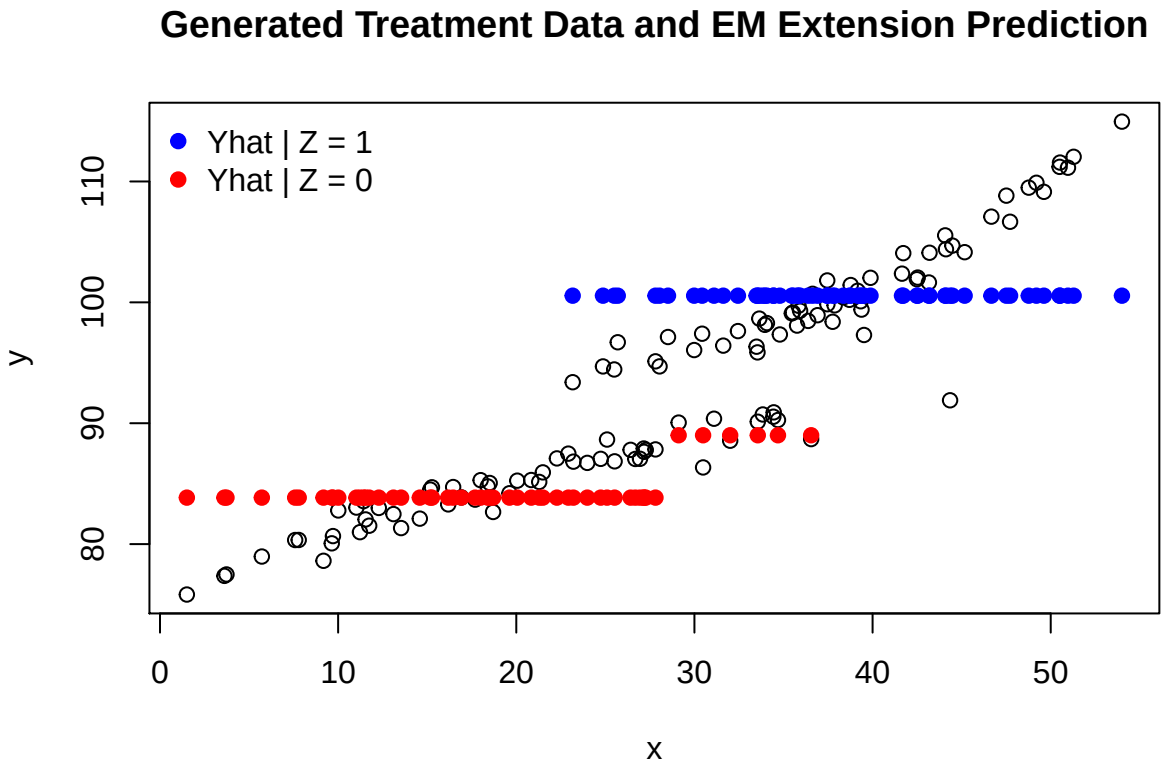
(iii) Update $\hat{\theta}^{(j+1)}$ and repeat until convergence at θ^*_μ

(iv) Generate $\omega_s \sim \mathcal{N}(\theta^*_\mu, \sigma^2_\mu)$ of size n

(v) Compute $\psi = \frac{1}{n}\sum_{i=1}^{n} 1(\omega_s > \bar{y})$

(vi) Create vector Z and set $Z_i = 1$ with probability ψ, else $Z_i = 0$

The EM algorithm above is repeated for each tree generated by the MH-Gibbs step. Every tree action (*grow, prune, change, swap*) is then informed by a new EM result, fully allowing the data to "speak for itself" without observing a treatment variable Z. The MH-Gibbs algorithm for each tree is largely unchanged in its description above. The only change is factoring the priors on ω and c into the likelihood computation of the MH step, which changes from $p(Y|X,T)$ to $p(Y|X,T,\omega)p(\omega)p(c)$. Setting a seed in the random number generator and using the same (Y,X) data for the BART replication plot above, the EM prediction results are plotted below (treating Z as completely unobservable) for a tree chain size of 2000.



References

Chipman, Hugh A, Edward I George, and Robert E McCulloch. 1998. "Bayesian Cart Model Search." *Journal of the American Statistical Association* 93 (443): 935–48.

Chipman, Hugh A, Edward I George, Robert E McCulloch, and others. 2010. "BART: Bayesian Additive Regression Trees." *The Annals of Applied Statistics* 4 (1): 266–98.

Hastie, Trevor, and Robert Tibshirani. 2000. "Bayesian Backfitting (with Comments and a Rejoinder by the Authors." *Statistical Science* 15 (3): 196–223.

Hill, Jennifer L. 2011. "Bayesian Nonparametric Modeling for Causal Inference." *Journal of Computational and Graphical Statistics* 20 (1): 217–40.

Supporting R Code

```r
# Bayesian Regression Tree - EM Extension
remove(list = objects())
library(tidyverse)

#### Generate data ---------------------------------
# data generation following Hill (2010)
set.seed(77)
p <- 0.5
z <- rbinom(120, 1, p)
y <- c()
x <- c()
for (i in 1:length(z)) {

  if (z[i] == 1) {

    x[i] <- rnorm(1, 40, 10)
    y[i] <- rnorm(1, 90 + exp(0.06*x[i]), 1)

  } else {

    x[i] <- rnorm(1, 20, 10)
    y[i] <- rnorm(1, 72 + 3*sqrt(x[i]), 1)

  }

}

#set.seed(NULL)

# rescale y to (-.5, .5) interval
ridx <- which(is.na(y))
y <- y[-ridx]
z <- z[-ridx]
x <- x[-ridx]
data <- data.frame(y = y,
                   x = x,
                   z = 0,
                   org_z = z)

#### Priors and likelihood -------------------------
# back to CGM98

# tree prior
p_split <- function(d, a = alpha, b = beta) {

  s <- a*(1 + d)^(-beta)
  # ru <- rbinom(1, 1, s)
  return(s)

}

# leaf param prior (mu)
```

```r
mus_draw <- function(n_tn, s2) {

  mus <- rnorm(n_tn, sd = sqrt(s2))
  return(mus)

}

# leaf param hyper
sig2_draw <- function(v = nu, lam = lambda) {

  s2 <- 1/rgamma(1, v/2, scale = v*lam/2)
  return(s2)

}

# log-likelihood
loglik <- function(n_vec, y_list, mu, n_tn, o_mle, ysbar, min_ys, max_ys,
                   s2 = sig2_mu, sc = a, v = nu, lam = lambda) {

  if (length(which(n_vec == 0)) > 0) {

    j <- which(n_vec == 0)
    y_list <- y_list[-j]
    n_vec <- n_vec[-j]
    n_tn <- n_tn - 1

  }

  s_vec <- map2_dbl(y_list, n_vec,
                    function(y, n) (n - 1)*var(y))
  # 1 obs in y_list -> NA for var(y), define as o
  s_vec[is.na(s_vec)] <- 0

  t_vec <- map2_dbl(y_list, n_vec,
                    function(y, n) ((n*sc)/(n + sc))*(mean(y) - mu)^2)

  f <- -sum(n_vec/2)*log(pi) + n_tn*v/2*log(v*lam) + n_tn/2*log(sc) -
    1/2*sum(log(n_vec + sc)) + sum(log(gamma((n_vec + v)/2))) -
    n_tn*log(gamma(v/2)) - sum((n_vec + v)/2)*sum(log(s_vec + t_vec + v*lam))

  val <- f*pnorm(ysbar, o_mle, sqrt(s2))*1/(max_ys - min_ys)

  return(val)

}

# Tree acceptance draw
rho_draw <- function(k_grow, k_prune, Lmp1, Lm, tdraw) {

  # if (Lmp1 < 0 & Lm > 0) {
  #
  #   l_rat <- 0
  #
```

```
  # } else if (Lmp1 > 0 & Lm < 0) {
  #
  #   l_rat <- 1
  #
  # } else if (Lmp1 == Inf & Lm == Inf) {
  #
  #   l_rat <- 1
  #
  # } else {
  #
  #   l_rat <- Lmp1/Lm
  #
  # }

  l_rat <- Lmp1/Lm

  if (tdraw == 'grow') {

    rho <- min(l_rat*k_prune, 1)

  } else if (tdraw == 'prune') {

    rho <- min(l_rat*1/k_grow, 1)

  } else {

    rho <- min(l_rat, 1)

  }

  rho <- ifelse(rho == Inf, 1, rho)
  bdraw <- rbinom(1, 1, rho)

  out <- list(rho = rho, draw = bdraw)
  return(out)

}

# Unobservable expectation
Ez <- function(theta_j, c, s2 = sig2_mu) {

  pdens <- 1/(sqrt(2*pi*s2))*exp(-1/(2*s2)*(c - theta_j)^2)
  cdens <- 1 - pnorm(c - theta_j, mean = theta_j, sd = sqrt(s2))
  return(theta_j + pdens/cdens)

}

# EM algorithm
EM_opt <- function() {

  err <- 1
  tol <- 1E-4
  itr <- 1
```

```
  max_itr <- 1E+4

  ys <- sample(y, length(y), replace = TRUE)
  ysbar <- mean(ys)
  a <- min(ys)
  b <- max(ys)
  c <- runif(1, a, b)
  n <- length(y)
  n_c <- length(y[y > c])
  theta_j <- mean(y)

  while (err > tol & itr <= max_itr) {

    theta_jp1 <- 1/n*(n_c*ysbar + (n - n_c)*Ez(theta_j, c))
    err <- abs(theta_jp1 - theta_j)
    itr <- itr + 1
    theta_j <- theta_jp1

    if (err < tol) {

      cat('EM Optimization',
          '\nTolerance met at iteration', itr,
          '\nMLE is', theta_jp1,
          '\n\n')

    }

    if (itr == max_itr) {

      cat('Tolerance condition not met. Exited algorithm at iteration', max_itr,
          '\n\n')

    }

  }

  out <- list(mle = theta_jp1, ysbar = ysbar,
              min_ys = a, max_ys = b)
  return(out)

}

omega_cond <- function() {

  EM <- EM_opt()
  ys_mle <- EM$mle
  ysbar <- EM$ysbar
  a <- EM$min_ys
  b <- EM$max_ys

  s <- rnorm(length(y), ys_mle, sig2_mu)
  delta <- length(s[s > mean(y)])/length(y)
  bdraw <- rbinom(1, 1, delta)
```

```r
  out <- list(bdraw = bdraw, mle = ys_mle, ysbar = ysbar,
              min_ys = a, max_ys = b)
  return(out)

}

#### Tree functions --------------------------------

# if action is grow
t_grow <- function() {

  s_direct <- sample(c('l', 'r'), 1, prob = c(1 - o_draw, o_draw))

  if(length(nodes_zl) == 0 | length(nodes_zr) == 0) {

    # define intial partition on z
    nodes_zl[[1]] <<- data %>%
      filter(z < split_z) %>%
      .$y
    nodes_zr[[1]] <<- data %>%
      filter(!y %in% nodes_zl[[1]]) %>%
      .$y

    # define initial depth
    i <- 1

  } else {

    # ifelse and if_else only handles conditions of length 1
    if (s_direct == 'l') {

      nodes <- nodes_zl
      nodes_r <- nodes_zlr
      splits <- splits_zl

    } else {

      nodes <- nodes_zr
      nodes_r <- nodes_zrr
      splits <- splits_zr

    }

    # new depth length to grow on
    i <- length(nodes) + 1

    # define uniform splitting rultes
    if (length(nodes) == 1) {

      splits[i] <- sample(data$x, 1)

    } else {
```

```
    splits[i] <- data$x %>%
      .[between(., min(.), splits[i - 1])] %>%
      sample(1)

  }

  split <- splits[i]

  # partition data at current split point for data in z <> .05
  split_y <- data %>%
    filter(x < split & y %in% nodes[[1]]) %>%
    .$y

  # check if minimum leaf obs is met
  if(length(split_y) >= min_leaf_obs) {

    # assign data to new node
    nodes[[i]] <- split_y

    # partition compliment data depending on node length
    if (length(nodes) <= 2) {

      split_comp_y <- data %>%
        filter(!y %in% nodes[[i]] &
                 y %in% nodes[[1]]) %>%
        .$y

    } else {

      split_comp_y <- data %>%
        filter(!y %in% unlist(nodes_r) &
                 !y %in% nodes[[i]] &
                 y %in% nodes[[1]]) %>%
        .$y

    }

    # assign data to compliment node
    nodes_r[[i]] <- split_comp_y

  } else {

    splits <- splits[-length(splits)]
    cat('Cannot partition.',
        '\nMinimum leaf obs of', min_leaf_obs, 'not met.',
        '\n')

  }

  if (s_direct == 'l') {

    nodes_zl <<- nodes
    nodes_zlr <<- nodes_r
```

```
      splits_zl <<- splits

    } else {

      nodes_zr <<- nodes
      nodes_zrr <<- nodes_r
      splits_zr <<- splits

    }

  }

  d <- ifelse(s_direct == 'l', length(nodes_zl), length(nodes_zr))
  out <- list(left_nodes = nodes_zl, right_nodes = nodes_zr,
              left_comps = nodes_zlr, right_comps = nodes_zrr,
              left_splits = splits_zl, right_splits = splits_zr,
              depth = d, direc = s_direct)
  return(out)

}

# if action is prune
t_prune <- function() {

  s_direct <- sample(c('l', 'r'), 1, prob = c(1 - o_draw, o_draw))

  if(length(nodes_zl) == 1  & s_direct == 'l') {

    cat('Cannot prune', s_direct, 'any further.',
        '\n')

  } else if (length(nodes_zr) == 1  & s_direct == 'r') {

    cat('Cannot prune', s_direct, 'any further.',
        '\n')

  } else {

    if (s_direct == 'l') {

      i <- length(nodes_zl)
      nodes_zl <<- nodes_zl[-i]
      nodes_zlr <<- nodes_zlr[-i]
      splits_zl <<- splits_zl[-i]

    } else {

      i <- length(nodes_zr)
      nodes_zr <<- nodes_zr[-i]
      nodes_zrr <<- nodes_zrr[-i]
      splits_zr <<- splits_zr[-i]

    }
```

```
  }

  d <- ifelse(s_direct == 'l', length(nodes_zl), length(nodes_zr))
  out <- list(left_nodes = nodes_zl, right_nodes = nodes_zr,
              left_comps = nodes_zlr, right_comps = nodes_zrr,
              left_splits = splits_zl, right_splits = splits_zr,
              depth = d, direc = s_direct)
  return(out)

}

# if action is change
t_change <- function() {

  s_direct <- sample(c('l', 'r'), 1, prob = c(1 - o_draw, o_draw))

  if(length(nodes_zl) == 1 & s_direct == 'l') {

    cat('Cannot change splitting rule for direction:', s_direct,
        '\nCurrent node is the initial node.',
        '\n')

  } else if (length(nodes_zr) == 1 & s_direct == 'r') {

    cat('Cannot change splitting rule for direction:', s_direct,
        '\nCurrent node is the initial node.',
        '\n')

  } else {

    if (s_direct == 'l') {

      nodes <- nodes_zl
      nodes_r <- nodes_zlr
      splits <- splits_zl

    } else {

      nodes <- nodes_zr
      nodes_r <- nodes_zrr
      splits <- splits_zr

    }

    # random node draw conditioned on random side (z <> 0.5)
    # specify change conditions based on node length
    if (length(nodes) > 2) {

      i <- sample(2:length(nodes), 1)

    } else {

      i <- 2
```

```
  }

  if (i == length(nodes)) {

    nodes <- nodes
    nodes_r <- nodes_r
    splits <- splits

  } else {

    nodes <- nodes[-c(i + 1:length(nodes))]
    nodes_r <- nodes_r[-c(i + 1:length(nodes_r))]
    splits <- splits[-c(i + 1:length(splits))]

  }

  # store current split
  old_split <- splits[i]

  # check split vector length to define new split
  if (i > 2) {

    splits[i] <- data$x %>%
      .[between(., min(.), splits[i - 1])] %>%
      sample(1)

  } else {

    splits[i] <- sample(data$x, 1)

  }

  split <- splits[i]

  # partition data at current split point for data in z <> .05
  split_y <- data %>%
    filter(x < split & y %in% nodes[[1]]) %>%
    .$y

  # check minimum leaf obs condition
  if(length(split_y) >= min_leaf_obs) {

    # assign data to new random node
    nodes[[i]] <- split_y

    # partition compliment data depending on node length
    if (length(nodes) <= 2) {

      split_comp_y <- data %>%
        filter(!y %in% nodes[[i]] &
                 y %in% nodes[[1]]) %>%
        .$y
```

```
    } else {

      split_comp_y <- data %>%
        filter(!y %in% unlist(nodes_r) &
                 !y %in% nodes[[i]] &
                 y %in% nodes[[1]]) %>%
        .$y

    }

    # assign data to new random compliment node
    nodes_r[[i]] <- split_comp_y

  } else {

    # iterate through new splitting rule until meeting obs const
    itr <- 1
    max_itr <- 20
    while (length(split_y) < min_leaf_obs & itr <= max_itr) {

      if (i > 2) {

        splits[i] <- data$x %>%
          .[between(., min(.), splits[i - 1])] %>%
          sample(1)

      } else {

        splits[i] <- sample(data$x, 1)

      }

      split <- splits[i]

      # partition data at current split point for data in z <> .05
      split_y <- data %>%
        filter(x < split & y %in% nodes[[1]]) %>%
        .$y

      cat('Finding valid splitting rule, iteration:', itr,
          '\n')
      itr <- itr + 1

      if (itr > max_itr) {

        # restore old split and partitioned data
        splits[i] <- old_split

        cat('Max splitting iterations reached. Original split kept.',
            '\n')

      }
```

```
      }

      # partition data based on splitting rule result
      split <- splits[i]
      split_y <- data %>%
        filter(x < split & y %in% nodes[[1]]) %>%
        .$y
      nodes[[i]] <- split_y

      # parition compliment data depending on node length
      if (length(nodes) <= 2) {

        split_comp_y <- data %>%
          filter(!y %in% nodes[[i]] &
                   y %in% nodes[[1]]) %>%
          .$y

      } else {

        split_comp_y <- data %>%
          filter(!y %in% unlist(nodes_r) &
                   !y %in% nodes[[i]] &
                   y %in% nodes[[1]]) %>%
          .$y

      }

    }

    if (s_direct == 'l') {

      nodes_zl <<- nodes
      nodes_zlr <<- nodes_r
      splits_zl <<- splits

    } else {

      nodes_zr <<- nodes
      nodes_zrr <<- nodes_r
      splits_zr <<- splits

    }

  }

  d <- ifelse(s_direct == 'l', length(nodes_zl), length(nodes_zr))
  out <- list(left_nodes = nodes_zl, right_nodes = nodes_zr,
              left_comps = nodes_zlr, right_comps = nodes_zrr,
              left_splits = splits_zl, right_splits = splits_zr,
              depth = d, direc = s_direct)
  return(out)

}
```

```
# if action is swap
t_swap <- function() {

  s_direct <- sample(c('l', 'r'), 1, prob = c(1 - o_draw, o_draw))

  if (length(nodes_zl) == 1 & s_direct == 'l') {

    cat('Cannot swap splitting rule for direction:', s_direct,
        '\nParent node is the initial node.',
        '\n')

  } else if (length(nodes_zr) == 1 & s_direct == 'r') {

    cat('Cannot swap splitting rule direction:', s_direct,
        '\nParent node is the initial node.',
        '\n')

  } else {

    if (s_direct == 'l') {

      nodes <- nodes_zl
      nodes_r <- nodes_zlr
      splits <- splits_zl

    } else {

      nodes <- nodes_zr
      nodes_r <- nodes_zrr
      splits <- splits_zr

    }

    # conditions for random parent node draw
    if (length(nodes) > 2) {

      i <- sample(2:length(nodes), 1)
      cat('Sample node', i, 'in length', length(nodes),
          '\n')

    } else {

      i <- 2

    }

    if (i == length(nodes)) {

      cat('There is no child node to swap with. Original nodes kept.',
          '\n')

    } else {
```

```
    # save child data to map to parent node (given symmetric child split rule)
    child_data <- nodes[[i + 1]]
    # subset nodes to i - 1 before parent to define child split rule on parent
    nodes <- nodes[-c(i:length(nodes))]
    nodes_r <- nodes_r[-c(i:length(nodes_r))]
    # swap splitting rule of parent i and child i + 1
    split <- splits[i + 1]
    splits <- splits[-c(i:length(splits))]
    splits[i] <- split

    # swap parent and child node data based on child swap rule
    nodes[[i]] <- child_data

    # partition new compliment data based on new parent node
    if (length(nodes) <= 2) {

      split_comp_y <- data %>%
        filter(!y %in% nodes[[i]] &
                 y %in% nodes[[1]]) %>%
        .$y

    } else {

      split_comp_y <- data %>%
        filter(!y %in% unlist(nodes_r) &
                 !y %in% nodes[[i]] &
                 y %in% nodes[[1]]) %>%
        .$y

    }

    # assign new compliment data based on child split
    nodes_r[[i]] <- split_comp_y

  }

  if (s_direct == 'l') {

    nodes_zl <<- nodes
    nodes_zlr <<- nodes_r
    splits_zl <<- splits

  } else {

    nodes_zr <<- nodes
    nodes_zrr <<- nodes_r
    splits_zr <<- splits

  }

}

d <- ifelse(s_direct == 'l', length(nodes_zl), length(nodes_zr))
```

```
  out <- list(left_nodes = nodes_zl, right_nodes = nodes_zr,
              left_comps = nodes_zlr, right_comps = nodes_zrr,
              left_splits = splits_zl, right_splits = splits_zr,
              depth = d, direc = s_direct)
  return(out)

}

#### Tree setup ----------------------------------

# initialize z partition nodes and split value
nodes_zl <- list()
nodes_zr <- list()
split_z <- 0.5

# initialize z_left and z_right nodes for > than conditions (go right)
nodes_zlr <- list(NA)
nodes_zrr <- list(NA)

# initialize split value vectors
splits_zl <- c(split_z)
splits_zr <- c(split_z)

# initialize leaf size constraint
min_leaf_obs <- 3

# choosen parameters -> CGM98 (alpha, beta, nu, lambda, a)
alpha <- 0.95
beta <- 2
nu <- 10
v_res <- var(resid(lm(y ~ x, data = data)))
lambda <- v_res
a <- v_res/var(y)

# chain length
m <- 2000

# initialize parameter matrix
leaf_params <- matrix(0, nrow = m, ncol = 15)

# initiate tree random action
tree_step <- c('grow', 'prune', 'change', 'swap')
draw_action <- function(d) {

  # kernel probabilities
  g <- p_split(d)
  p <- (1 - g)/3
  draw <- sample(tree_step, 1, prob = c(g, rep(p, length(tree_step) - 1)))
  out <- list(g_kern = g, p_kern = p, draw = draw)
  return(out)

}
```

```r
# get tree data
get_treedata <- function(i, s2 = sig2_mu) {

  # get tree information
  y_list <- c(nodes_zl[length(nodes_zl)], nodes_zlr[-1],
              nodes_zr[length(nodes_zr)], nodes_zrr[-1])

  n_tnl <- 1 + length(nodes_zlr[-1])
  n_tnr <- 1 + length(nodes_zrr[-1])
  n_tns <- n_tnl + n_tnr
  n_vec <- sapply(y_list, length)

  l_splits <- splits_zl
  r_splits <- splits_zr

  params <- rnorm(n_tns, mean(y), sqrt(s2))
  mu <- mean(params)
  leaf_params[i, 1:n_tns] <<- sapply(y_list, mean)

  out <- list(y_list = y_list, n_tns = n_tns,
              n_tnl = n_tnl, n_tnr = n_tnr, n_vec = n_vec,
              l_splits = l_splits, r_splits = r_splits,
              params = params, mu = mu)
  return(out)

}

#### MCMC posterior generation ----------------------------

# initial values for tree generation
sig2_mu <- sig2_draw()/a
omega <- omega_cond()
o_draw <- omega$bdraw
o_mle <- omega$mle
o_ysbar <- omega$ysbar
o_min_ys <- omega$min_ys
o_max_ys <- omega$max_ys
data[which(data$y >= o_mle), 'z'] <- 1

# initial tree and likelihood
tree_m <- t_grow()
d <- tree_m$depth
treedata_m <- get_treedata(1)
y_list <- treedata_m$y_list
n_tn <- treedata_m$n_tns
n_vec <- treedata_m$n_vec
mu <- treedata_m$mu
loglik_m <- loglik(n_vec, y_list, mu, n_tn,
                   o_mle, o_ysbar, o_min_ys, o_max_ys)

# initialize tree data storage
tree_list <- list(tree_m)
treedata_list <- list(treedata_m)
```

```
z_vals <- matrix(0, nrow = length(data$z), ncol = m)
z_vals[, 1] <- data$z

o_mle_vec <- o_mle
o_draw_vec <- o_draw
rho_bin_vec <- NA
rho_val_vec <- NA

# MH implementation
for (i in 2:m) {

  # draw action
  tree_action <- draw_action(d)
  draw <- tree_action$draw

  # omega condition
  sig2_mu <- sig2_draw()/a
  omega <- omega_cond()
  o_draw <- omega$bdraw
  o_mle <- omega$mle
  o_ysbar <- omega$ysbar
  o_min_ys <- omega$min_ys
  o_max_ys <- omega$max_ys

  o_mle_vec[i] <- o_mle
  o_draw_vec[i] <- o_draw

  data[which(data$y >= o_mle), 'z'] <- 1
  z_vals[, i] <- data$z

  if (draw == 'grow') {

    tree_mp1 <- t_grow()
    g_kern <- tree_action$g_kern
    p_kern <- tree_action$p_kern

  } else if (draw == 'prune') {

    tree_mp1 <- t_prune()
    g_kern <- tree_action$g_kern
    p_kern <- tree_action$p_kern

  } else if (draw == 'change') {

    tree_mp1 <- t_change()

  } else {

    tree_mp1 <- t_swap()

  }

  # get new tree data
```

```
  treedata_mp1 <- get_treedata(i)
  y_list <- treedata_mp1$y_list
  n_tn <- treedata_mp1$n_tns
  n_vec <- treedata_mp1$n_vec
  mu <- treedata_mp1$mu
  loglik_mp1 <- loglik(n_vec, y_list, mu, n_tn,
                       o_mle, o_ysbar, o_min_ys, o_max_ys)

  # tree_mp1 acceptance parameter
  rho <- rho_draw(g_kern, p_kern, loglik_mp1, loglik_m, draw)
  rho_bin_vec[i] <- rho$draw
  rho_val_vec[i] <- rho$rho

  if (rho$draw == 1) {

    # T* = tree_mp1, store tree_mp1 data from above
    # store tree_mp1 as tree_m
    tree_list[[i]] <- tree_mp1
    treedata_list[[i]] <- treedata_mp1

    # store updated likelihood value as mth iter likelihood
    loglik_m <- loglik_mp1

    cat('T(m+1) = T*',
        '\n')

  } else {

    # T* = tree_m
    tree_list[[i]] <- tree_m
    treedata_list[[i]] <- treedata_list[[i - 1]]

    # cancel drawn parameters for unaccepted tree
    leaf_params[i, ] <- leaf_params[i - 1, ]

    cat('T(m+1) = T(m)',
        '\n')

  }

  tree_m <- tree_list[[i]]
  d <- tree_m$depth

  cat('------Bayes tree draw action:', draw,
      '\n------Iteration', i, '/', m,
      '\n\n')
}

#### Prediction ----------------------------

# weighted nodes expectation
vis_nodes <- data.frame(l = sapply(tree_list, function(x) length(x$left_splits)),
                        r = sapply(tree_list, function(x) length(x$right_splits)))
```

```r
get_Enodes <- function(direc) {

  w_node <- sum(table(vis_nodes[[direc]])/m*sort(unique(vis_nodes[[direc]])))
  idx <- which.min(abs(w_node - unique(vis_nodes[[direc]])))
  Enode <- unique(vis_nodes[[direc]])[idx]

  return(Enode)

}

Enodes <- c(get_Enodes('l'), get_Enodes('r'))

# expected left splits
sub_lsplits_idx <- sapply(tree_list, function(x) length(x$left_splits) == Enodes[1])
sub_lsplits <- sapply(tree_list[c(sub_lsplits_idx)], function(x) x$left_splits)
if (Enodes[1] > 1) {

  # ifelse not handling return rowMeans()
  Elsplits <- rowMeans(sub_lsplits)

} else {

  Elsplits <- split_z

}

# expected right splits
sub_rsplits <- sapply(tree_list, function(x) length(x$right_splits) == Enodes[2])
sub_rsplits <- sapply(tree_list[c(sub_rsplits)], function(x) x$right_splits)
if (Enodes[2] > 1) {

  Ersplits <- rowMeans(sub_rsplits)

} else {

  Ersplits <- split_z

}

# get associated leaf parameters
z <- round(rowMeans(z_vals))
data$z <- z

get_leafparams <- function(len_sp, direc) {

  p_vec <- c()

  if (direc == 'right') {

    for (i in 1:len_sp) {

      if (len_sp == 1) {
```

```
      p_vec[i] <- data %>%
        filter(z > Ersplits[1]) %>%
        .$y %>%
        mean()

    } else if (i == len_sp) {

      p_vec[i] <- data %>%
        filter(z > Ersplits[1] &
                 x < Ersplits[i]) %>%
        .$y %>%
        mean()

    } else if (i == 1) {

      p_vec[i] <- data %>%
        filter(z > Ersplits[1] &
                 x > Ersplits[i + 1]) %>%
        .$y %>%
        mean()

    } else {

      p_vec[i] <- data %>%
        filter(z > Ersplits[1] &
                 x < Ersplits[i] &
                 x > Ersplits[i + 1]) %>%
        .$y %>%
        mean()

    }

  }

} else {

  for (i in 1:len_sp) {

    if (len_sp == 1) {

      p_vec[i] <- data %>%
        filter(z < Elsplits[1]) %>%
        .$y %>%
        mean()

    } else if (i == len_sp) {

      p_vec[i] <- data %>%
        filter(z < Elsplits[1] &
                 x < Elsplits[i]) %>%
        .$y %>%
        mean()
```

```
      } else if (i == 1) {

        p_vec[i] <- data %>%
          filter(z < Elsplits[1] &
                   x > Elsplits[i + 1]) %>%
          .$y %>%
          mean()

      } else {

        p_vec[i] <- data %>%
          filter(z < Elsplits[1] &
                   x < Elsplits[i] &
                   x > Elsplits[i + 1]) %>%
          .$y %>%
          mean()

      }

    }

  }

  return(p_vec)

}

# retrieve parameter values
Elparams <- get_leafparams(length(Elsplits), 'left')
Erparams <- get_leafparams(length(Ersplits), 'right')

# splits may be too close together and produce no obs
if (any(is.na(Elparams))) {

  naidx <- which(is.na(Elparams))
  Elparams <- Elparams[-naidx]
  Elsplits <- Elsplits[-(naidx + 1)]

} else if (any(is.na(Erparams))) {

  naidx <- which(is.na(Erparams))
  Erparams <- Erparams[-naidx]
  Ersplits <- Ersplits[-(naidx + 1)]

}

# set values for plotting fits
data$yfit <- 0
llparams <- length(Elparams)
lrparams <- length(Erparams)

if (llparams == 1) {
```

```
  idx <- which(data$z < Elsplits[1])
  data$yfit[idx] <- Elparams[1]

}

if (llparams == 2) {

  idx1 <- which(data$z < Elsplits[1] &
                data$x >= Elsplits[2])
  data$yfit[idx1] <- Elparams[1]
  idx2 <- which(data$z < Elsplits[1] &
                data$x < Elsplits[2])
  data$yfit[idx2] <- Elparams[2]

}

for (i in 1:llparams) {

  if (i == 1) {

    idx <- which(data$z < Elsplits[1] &
                 data$x >= Elsplits[2])
    data$yfit[idx] <- Elparams[1]

  } else {

    idx <- which(data$z < Elsplits[1] &
                 data$x < Elsplits[i] &
                 data$x >= Elsplits[i + 1])
    data$yfit[idx] <- Elparams[i]

  }

  if (i == llparams) {

    idx <- which(data$z < Elsplits[1] &
                 data$x < Elsplits[i])
    data$yfit[idx] <- Elparams[i]

  }

}

if (lrparams == 1) {

  idx <- which(data$z > Ersplits[1])
  data$yfit[idx] <- Erparams[1]

}

if (lrparams == 2) {

  idx1 <- which(data$z > Ersplits[1] &
```

```
                  data$x >= Ersplits[2])
  data$yfit[idx1] <- Erparams[1]
  idx2 <- which(data$z > Ersplits[1] &
                  data$x < Ersplits[2])
  data$yfit[idx2] <- Erparams[2]

}

for (i in 1:lrparams) {

  if (i == 1) {

    idx <- which(data$z > Ersplits[1] &
                   data$x >= Ersplits[2])
    data$yfit[idx] <- Erparams[1]

  } else {

    idx <- which(data$z > Ersplits[1] &
                   data$x < Ersplits[i] &
                   data$x >= Ersplits[i + 1])
    data$yfit[idx] <- Erparams[i]

  }

  if (i == lrparams) {

    idx <- which(data$z > Ersplits[1] &
                   data$x < Ersplits[i])
    data$yfit[idx] <- Erparams[i]

  }

}

# plot results
EM_ext_plot <- function() {

  plot(data$x, data$y, xlab = 'x', ylab = 'y')
  points(data[data$z < 0.5, 'x'], data[data$z < 0.5, 'yfit'],
         col = 'red', pch = 19, cex = 1)
  points(data[data$z > 0.5, 'x'], data[data$z > 0.5, 'yfit'],
         col = 'blue', pch = 19, cex = 1)
  title(main = 'Generated Treatment Data and EM Extension Prediction')
  legend("topleft", pch = 19, col = c('blue', 'red'), box.lty = 0,
         legend = c('Yhat | Z = 1', 'Yhat | Z = 0'))

}
```