Project 5

Benjamin Draves

```r
#------------------------------------------------
# Neural Network Metropolis-within-Gibbs Sampler
#------------------------------------------------

# helper functions
LOGEPS <- log(.Machine$double.eps/2)
log1pe <- function(x) {
    # vectorized version: `x` can be a vector
    l <- ifelse(x > 0, x, 0)  # shift
    x <- ifelse(x > 0, -x, x)  # range reduction: `x = -abs(x)`
    ifelse(x < LOGEPS, l, l + log(1 + exp(x)))
}

# ft: the neural network 'function' - wrapper function
# arguments: 1. Data vector x 2. Model paramter vector a0 2.
# Model paramter matrix a (ak in the rows of a) 2. Model
# paramter vector b
ft <- function(x, a0, a, b) {
    K <- nrow(a)
    tot <- 0
    for (k in 1:K) {
        tot <- tot + b[k] * tanh(a0[k] + sum(a[, k] * x))
    }
    return(tot)
}

# f: the full neural network - wrapper function arguments: 1.
# Data matrix x 2. Model paramter vector a0 2. Model paramter
# matrix a (ak in the rows of a) 2. Model paramter vector b
f <- function(X, a0, a, b) apply(X, 1, function(x) ft(x, a0,
    a, b))

# logpi: A function that evaluates log[pi(*|theta\j, Y, X)]
# arguments: 1. Response vector Y 2. Design matrix X 3. Fixed
# standard deviation c^2 4. The current paramter value theta
# 5. Parameter values (a0, a, b)

logpi <- function(Y, X, c2, theta, a0, a, b) {
    eval <- f(X, a0, a, b)

    if (sum(is.na(eval)) != 0) {
        print("STOPPED AT logpi")
        break
    }

    -(theta)^2/(2 * c2) + sum(Y * eval) - sum(log1pe(eval))
}

# gibbs_sampler: a function that returns the samples from the
# MCMC arguments: 1. Response vector Y 2. Design matrix X 3.
# Initial parmater value matrix theta0 (K, 2 + p) - (a_0k,
# a_k, b_k) 4. K - depth of NN (default K = 10) 5. Initial
# 'step sizes' matrix v (K, p + 2) 6. Learing rate function
# gamma: t --> 1/t^gamma (default = 2) 7. Desired acceptance
# rate r (default = 30%) 8. Maximum number of iterations
# (default = 5000)
```

```r
gibbs_sampler <- function(Y, X, theta0, v, K = 10, gamma = 2,
    c2 = 1, r = 0.3, max.iter = 5000) {

    #------------------------------------------
    # set up data structures - paramter values
    #------------------------------------------
    p <- ncol(X)
    tfin <- NA

    # Intercept term
    a0 <- matrix(NA, nrow = K, ncol = max.iter)
    a0[, 1] <- theta0[, 1]

    # Weights
    b <- matrix(NA, nrow = K, ncol = max.iter)
    b[, 1] <- theta0[, p + 2]

    # Slope terms
    a <- array(NA, dim = c(K, p, max.iter))
    a[, , 1] <- theta0[, 2:(p + 1)]

    #------------------------------------------
    # set up data structures - step size
    #------------------------------------------
    v0 <- matrix(NA, nrow = K, ncol = max.iter)
    v0[, 1] <- v[, 1]

    vb <- matrix(NA, nrow = K, ncol = max.iter)
    vb[, 1] <- v[, p + 2]

    va <- array(NA, dim = c(K, p, max.iter + 1))
    va[, , 1] <- v[, 2:(p + 1)]

    #------------------------------------------
    # set up data structures - acceptance 0/1
    #------------------------------------------
    acceptance <- array(NA, dim = c(K, p + 2, max.iter))

    # begin samples
    for (t in 2:max.iter) {
        #------------------------------------------------------
        # Update parameters
        #------------------------------------------------------

        # copy over paramaters for efficency
        a0[1:K, t] <- a0[1:K, t - 1]
        b[1:K, t] <- b[1:K, t - 1]
        a[, , t] <- a[, , t - 1]

        # update a0
        for (k in 1:K) {
            # propose new sample N(a0[k,t-1],v0[k,t-1])
            tstar <- rnorm(1, a0[k, t - 1], v0[k, t - 1])

            # calculate log acceptance probability
            top <- a0[1:K, t]
            top[k] <- tstar
            logp <- min(c(0, logpi(Y, X, c2, tstar, top, a[,
                , t], b[, t]) - logpi(Y, X, c2, a0[k, t], a0[,
                t], a[, , t], b[, t])))
            # Update new value
            a0[k, t] <- ifelse(log(runif(1)) < logp, tstar, a0[k,
                t])

            # Update step size
            v0[k, t] <- v0[k, t - 1] + 1/(t^(gamma)) * (exp(logp) -
                r)

            # update acceptance
            acceptance[k, 1, t - 1] <- ifelse(tstar == a0[k,
                t], 1, 0)
        }

        # update b
        for (k in 1:K) {
            # propose new sample N(b[k, i, t-1],vb[k, i, t-1])
            tstar <- rnorm(1, b[k, t - 1], vb[k, t - 1])

            # calculate log acceptance probability
            top <- b[1:K, t]
            top[k] <- tstar
            logp <- min(c(0, logpi(Y, X, c2, tstar, a0[, t],
                a[, , t], top) - logpi(Y, X, c2, b[k, t], a0[,
                t], a[, , t], b[, t])))

            # Update new value
            b[k, t] <- ifelse(log(runif(1)) < logp, tstar, b[k,
                t])

            # Update step size
            vb[k, t] <- vb[k, t - 1] + 1/(t^(gamma)) * (exp(logp) -
                r)

            # update acceptance
            acceptance[k, p + 2, t - 1] <- ifelse(tstar == b[k,
                t], 1, 0)
        }

        # update a
        for (k in 1:K) {
            for (i in 1:p) {
                # propose new sample N(a[k, i, t-1],va[k, i, t-1])
                tstar <- rnorm(1, a[k, i, t - 1], va[k, i, t -
                  1])

                # calculate log acceptance probability
                top <- a[, , t]
                top[k, i] <- tstar
                logp <- min(c(0, logpi(Y, X, c2, tstar, a0[,
                  t], top, b[, t]) - logpi(Y, X, c2, a[k, i,
                  t - 1], a0[, t], a[, , t], b[, t])))

                # Update new value
                a[k, i, t] <- ifelse(log(runif(1)) < logp, tstar,
                  a[k, i, t - 1])

                # Update step size
                va[k, i, t] <- va[k, i, t - 1] + 1/(t^(gamma)) *
                  (exp(logp) - r)
```

```
                # update acceptance
                acceptance[k, i + 1, t - 1] <- ifelse(tstar ==
                  a[k, i, t], 1, 0)

            }
        }
        #-------------------------------------------------------
        # Check Convergence
        #-------------------------------------------------------
        print(t)
        if (t == max.iter) {
            tfin <- max.iter
            break
        }

    }

    # format results
    res <- array(NA, dim = c(K, p + 2, tfin))
    res[, 1, ] <- a0
    res[, 2:(p + 1), ] <- a
    res[, p + 2, ] <- b

    # return(list(parameters = res, acceptance = acceptance))
    return(parameters = res)

}


#------------------------------------------------
# Parallel Tempering algorithm
#------------------------------------------------
# update_paramter: function that completes an update for PT
# parameter arguments: 1. Y: response vector (n x 1) 2. X:
# covariate matrix (n x p) 3. c2: numeric prior variance 4.
# theta: current value of parameters in each chain (d x 1 )
# 5. params: most recent parameters of full model (K x p+2)
# 6. vec: which element we are updating in params (2 x 1) 7.
# temps: vector of temperatures (d x 1) 8. stepsize: numeric
# step size

update_parameter <- function(Y, X, c2, theta, params, vec, temps,
    step_size) {

    # set up local variables
    d <- length(temps)
    p <- ncol(X)
    theta_bar <- numeric(d)
    lprobs <- numeric(d)


    for (ell in 1:d) {
        # update each chain
        tstar <- rnorm(1, mean = theta[ell], sd = step_size)

        # calculate log acceptance probability
        top <- params
        top[vec[1], vec[2]] <- tstar
        logp <- 1/temps[ell] * min(c(0, logpi(Y, X, c2, tstar,
            top[, 1], top[, 2:(p + 1)], top[, p + 2]) - logpi(Y,
            X, c2, theta[ell], params[, 1], params[, 2:(p + 1)],
```

```r
            params[, 2 + p])))

        # store log acceptance prob
        lprobs[ell] <- logp

        # update estimate
        theta_bar[ell] <- ifelse(log(runif(1)) < logp, tstar,
            theta[ell])

    }
    # draw I in (1, d-1)
    I <- sample(1:(d - 1), 1)

    # calculate paramter matrix with theta_bar[I]/theta_bar[I+1]
    top <- params
    top[vec[1], vec[2]] <- theta_bar[I]
    bottom <- params
    bottom[vec[1], vec[2]] <- theta_bar[I + 1]

    # caclulate log alpha
    diff_h <- logpi(Y, X, c2, theta_bar[I], top[, 1], top[, 2:(p +
        1)], top[, p + 2]) - logpi(Y, X, c2, theta_bar[I + 1],
        bottom[, 1], bottom[, 2:(p + 1)], bottom[, p + 2])
    logalpha <- min(c(0, (1/(temps[I + 1]) - 1/(temps[I])) *
        diff_h))

    # swap /no swap
    if (log(runif(1)) < logalpha) {
        # if yes swap
        tmp <- theta_bar[I]
        theta_bar[I] <- theta_bar[I + 1]
        theta_bar[I + 1] <- tmp
    }

    # return update values of each chain
    return(list(parameter = theta_bar[1], positions = theta_bar[-1],
        lprobs = lprobs))

}


# update_paramter: function that completes an update for PT
# parameter arguments: 1. Y: response vector (n x 1) 2. X:
# covariate matrix (n x p) 3. theta0: initial parameter value
# matrix (K x p+2) 4. v0: initial step size matrix (K x p+2)
# 5. temps: initial temperatures vector (d x 1) 6. K: neural
# network depth K 7. c2: prior variance, default 1 8.
# max.iter: maximum number of iterations, defaul 5000 Used
# for adaptive step size 9. gamma: learning rate (.5, infty),
# default INFTY (no adaptive updates) 10. r: desired
# acceptance probability, default .3
pt_sampler <- function(Y, X, theta0, v0, temps, K = 10, c2 = 1,
    max.iter = 5000, gamma = Inf, r = 0.3) {

    # set up local varibles
    p <- ncol(X)
    d <- length(temps)


    # set up storage for parallel chains
    pt_chains_positions <- array(1, dim = c(K, p + 2, d - 1))
    # set up storage for parameter vectors
    theta <- array(1, dim = c(K, p + 2, max.iter))
    theta[, , 1] <- theta0

    # set up storage for stepsizes
    v <- array(1, dim = c(K, p + 2, max.iter))
    v[, , 1] <- v0


    for (t in 2:max.iter) {

        # copy over past parameters for computational convience
        theta[, , t] <- theta[, , t - 1]

        for (k in 1:K) {
            for (j in 1:(p + 2)) {
                # get current placement of chains
                theta_here <- c(theta[k, j, t], pt_chains_positions[k,
                  j, ])

                # update parameters
                tmp <- update_parameter(Y, X, c2, theta_here,
                  theta[, , t], c(k, j), temps, v[k, j, t - 1])
                theta[k, j, t] <- tmp$parameter
                pt_chains_positions[k, j, ] <- tmp$positions

                # update step size
                v[k, j, t] <- v[k, j, t - 1]  #+ 1/(t^(gamma)) * (exp(tmp$lprobs[1]) - r)

            }
        }
        print(t)
    }

    return(theta)


}
```