bfastmonitor for dummies

Christopher Stephan

09/17/2014

```
function (data, start, formula = response ~ trend + harmon, order = 3,
    lag = NULL, slag = NULL, history = c("ROC", "BP", "all"),
    type = "OLS-MOSUM", h = 0.25, end = 10, level = 0.05, hpc = "none",
    verbose = FALSE, plot = FALSE)
{
    # Replicates the level value two times
    level <- rep(level, length.out = 2)

    # If the time series (data) is not a ts object it is coerced to such
    if (!is.ts(data))
        data <- as.ts(data)

    # Returns the number of samples per unit time and deltat the time
    # interval between observation.
    freq <- frequency(data)

    # Function to convert parameter x to a numeric time stamp when the start
    # parameter of bfastmonitor() is passed as pair of period / cycle.  For
    # a time series of frequency f, time n+i/f is presented as c(n, i+1).
    # For further understanding execute ?start and see details.

    # L is used as a suffix to qualify any number with the intent of making
    # it an explicit integer. It is useed to get the code to run faster and
    # consume less memory. A double ("numeric") vector uses 8 bytes per
    # element. An integer vector uses only 4 bytes per element.
    time2num <- function(x) if (length(x) > 1L)
        x[1L] + (x[2L] - 1)/freq
    else x

    # Initialize the starting date of the monitoring period.
    start <- time2num(start)

    # Dataframe for subsequent regression modeling.
    data_tspp <- bfastpp(data, order = order, lag = lag, slag = slag)

    # Building a subset from data_tspp containing only the data until the
    # beginning of the monitoring period (the history data).
    history_tspp <- subset(data_tspp, time < start)

    # If the passed history parameter is null extract the times the first
    # observations was taken.
    if (is.null(history)) {
        # dont understand because history_tspp is a data frame and no time
        # series, nor a vector or matrix
        history <- start(history_tspp$response)
    }

    # If passed history parameter is a character.
```

```
    else if (all(is.character(history))) {
        # Matches the perhaps abbreviated passed history parameter against
        # possible parameter values that are obtained from the the signature
        # of the function from which match.arg is called (-> bfastmonitor()).
        history <- match.arg(history)
        # Switch evaluates the history parameter and accordingly executes
        # one of the further functions. The returned restult is stored back to
        # the history variable.
        history <- switch(history,
            all = start(history_tspp$response),
            ROC = history_roc(formula, data = history_tspp, level = level[2]),
            BP = history_break(formula, data = history_tspp, hpc = hpc)
        )
    }
    else if (all(is.function(history))) {
        history <- history(formula, data = history_tspp)
    }
    history <- time2num(history)
    history_tspp <- subset(history_tspp, time >= history)
    if (verbose) {
        cat("\nBFAST monitoring\n\n1. History period\n")
        cat(sprintf("Stable period selected: %i(%i)--%i(%i)\n",
            start(history_tspp$response)[1], start(history_tspp$response)[2],
            end(history_tspp$response)[1], end(history_tspp$response)[2]))
        cat(sprintf("Length (in years): %f\n", NROW(history_tspp)/freq))
    }
    test_tspp <- history_tspp
    test_mefp <- mefp(formula, data = test_tspp, type = type,
        period = end, h = h, alpha = level[1])
    test_lm <- lm(formula, data = test_tspp)
    if (floor(h * NROW(test_tspp)) <= 1 | NROW(test_tspp) <=
        length(coef(test_lm))) {
        ok <- FALSE
        warning("too few observations in selected history period")
    }
    else {
        ok <- TRUE
    }
    if (verbose) {
        cat("Model fit:\n")
        print(coef(test_lm))
    }
    test_tspp <- subset(data_tspp, time >= history)
    if (ok) {
        test_mon <- monitor(test_mefp, data = test_tspp, verbose = FALSE)
        tbp <- if (is.na(test_mon$breakpoint))
            NA
        else test_tspp$time[test_mon$breakpoint]
        if (verbose) {
            cat("\n\n2. Monitoring period\n")
            cat(sprintf("Monitoring starts at: %i(%i)\n", floor(start),
                round((start - floor(start)) * freq) + 1))
            if (is.na(tbp)) {
```

```
                cat("Break detected at: -- (no break)\n\n")
            }
            else {
                cat(sprintf("Break detected at: %i(%i)\n\n",
                  floor(tbp), round((tbp - floor(tbp)) * freq) +
                    1))
            }
        }
    }
    else {
        test_mon <- NA
        tbp <- NA
    }
    if (ok) {
        test_tspp$prediction <- predict(test_lm, newdata = test_tspp)
        new_data <- subset(test_tspp, time >= start)
        magnitude <- median(new_data$response - new_data$prediction,
            na.rm = TRUE)
    }
    else {
        test_tspp$prediction <- NA
        magnitude <- NA
    }
    rval <- list(data = data, tspp = test_tspp, model = test_lm,
        mefp = test_mon, history = c(head(history_tspp$time,
            1), tail(history_tspp$time, 1)), monitor = c(start,
            tail(test_tspp$time, 1)), breakpoint = tbp, magnitude = magnitude)
    class(rval) <- "bfastmonitor"
    if (plot)
        plot(rval)
    return(rval)
}
```

```
########################################
## Reversely Ordered CUSUM (ROC) test ##
########################################

## A technique to verify whether or not the historical period is stable or not
## reversely order sample and perform recursive CUSUM test
history_roc <- function(formula, data, level = 0.05) {
  # Number of rows in the data (historical period).
  n <- nrow(data)
  # Storing reverse ordering of the data in data_rev.
  data_rev <- data[n:1,]
  # Making a time series of the response variables.
  data_rev$response <- ts(data_rev$response)

  # efp will return a one-dimensional empirical process of sums of residuals.
  y_rcus <- efp(formula, data = data_rev, type = "Rec-CUSUM")

  y_start <- if(sctest(y_rcus)$p.value < level) {
    length(y_rcus$process) - min(which(abs(y_rcus$process)[-1] > boundary(y_rcus)[-1])) + 1
  } else {
```

```
    1
  }
  data$time[y_start]
}
```

```
###################################
## Bai & Perron last breakpoint ##
###################################

history_break <- function(formula, data, h = NULL, hpc = "none") {
  n <- nrow(data)
  ## rule of thumb for minimal segment size
  if(is.null(h)) h <- 6 * NCOL(model.matrix(formula, data = data[0,]))

  ## conduct breakpoints estimation
  bp <- breakpoints(formula, data = data, h = h, hpc = hpc)

  y_start <- tail(breakpoints(bp)$breakpoints, 1)
  y_start <- if(is.na(y_start)) 1 else y_start + 1
  data$time[y_start]
}
```