```
### hyper parameters for P-splines baselearner (including tensor product P-splines)
hyper_bbs <- function(mf, vary, knots = 20, boundary.knots = NULL, degree = 3,
                      differences = 2, df = 4, lambda = NULL, center = FALSE,
                      cyclic = FALSE, constraint = "none", deriv = 0L) {

  knotf <- function(x, knots, boundary.knots) {
    if (is.null(boundary.knots))
      boundary.knots <- range(x, na.rm = TRUE)
    ## <fixme> At the moment only NULL or 2 boundary knots can be specified.
    ## Knot expansion is done automatically on an equidistand grid.</fixme>
    if ((length(boundary.knots) != 2) || !boundary.knots[1] < boundary.knots[2])
      stop("boundary.knots must be a vector (or a list of vectors) ",
           "of length 2 in increasing order")
    if (length(knots) == 1) {
      knots <- seq(from = boundary.knots[1],
                   to = boundary.knots[2], length = knots + 2)
      knots <- knots[2:(length(knots) - 1)]
    }
    list(knots = knots, boundary.knots = boundary.knots)
  }
  nm <- colnames(mf)[colnames(mf) != vary]
  if (is.list(knots)) if(!all(names(knots) %in% nm))
    stop("variable names and knot names must be the same")
  if (is.list(boundary.knots)) if(!all(names(boundary.knots) %in% nm))
    stop("variable names and boundary.knot names must be the same")
  if (!identical(center, FALSE) && cyclic)
    stop("centering of cyclic covariates not yet implemented")
  ret <- vector(mode = "list", length = length(nm))
  names(ret) <- nm
  for (n in nm)
    ret[[n]] <- knotf(mf[[n]], if (is.list(knots)) knots[[n]] else knots,
                      if (is.list(boundary.knots)) boundary.knots[[n]]
                      else boundary.knots)
  if (cyclic & constraint != "none")
    stop("constraints not implemented for cyclic B-splines")
  stopifnot(is.numeric(deriv) & length(deriv) == 1)
  ## prediction is usually set in/by newX()
  list(knots = ret, degree = degree, differences = differences,
       df = df, lambda = lambda, center = center, cyclic = cyclic,
       Ts_constraint = constraint, deriv = deriv, prediction = FALSE)
}

### model.matrix for P-splines baselearner (including tensor product P-splines)
```

```
X_bbs <- function(mf, vary, args) {

  stopifnot(is.data.frame(mf))
  mm <- lapply(which(colnames(mf) != vary), function(i) {
    if (!args$cyclic) {
      X <- bsplines(mf[[i]],
                    knots = args$knots[[i]]$knots,
                    boundary.knots = args$knots[[i]]$boundary.knots,
                    degree = args$degree,
                    Ts_constraint = args$Ts_constraint,
                    deriv = args$deriv, extrapolation = args$prediction)
    } else { ## if cyclic spline
      X <- cbs(mf[[i]],
               knots = args$knots[[i]]$knots,
               boundary.knots = args$knots[[i]]$boundary.knots,
               degree = args$degree,
               deriv = args$deriv)
    }
    class(X) <- "matrix"
    return(X)
  })
  ### options
  MATRIX <- any(sapply(mm, dim) > c(500, 50)) || (length(mm) > 1)
  MATRIX <- MATRIX && options("mboost_useMatrix")$mboost_useMatrix
  if (MATRIX) {
    diag <- Diagonal
    for (i in 1:length(mm)){
      tmp <- attributes(mm[[i]])[c("degree", "knots", "Boundary.knots")]
      mm[[i]] <- Matrix(mm[[i]])
      attributes(mm[[i]])[c("degree", "knots", "Boundary.knots")] <- tmp
    }
  }
  if (length(mm) == 1) {
    X <- mm[[1]]
    if (vary != "") {
      by <- model.matrix(as.formula(paste("~", vary, collapse = "")),
                         data = mf)[ , -1, drop = FALSE] # drop intercept
      DM <- lapply(1:ncol(by), function(i) {
        ret <- X * by[, i]
        colnames(ret) <- paste(colnames(ret), colnames(by)[i], sep = ":")
        ret
      })
```

```r
    X <- do.call("cbind", DM)
  }
  if (args$differences > 0){
    if (!args$cyclic) {
      K <- diff(diag(ncol(mm[[1]])), differences = args$differences)
    } else {
      ## cyclic P-splines
      differences <- args$differences
      K <- diff(diag(ncol(mm[[1]]) + differences),
                differences = differences)
      tmp <- K[,(1:differences)]   # save first "differences" columns
      K <- K[,-(1:differences)]    # drop first "differences" columns
      indx <- (ncol(mm[[1]]) - differences + 1):(ncol(mm[[1]]))
      K[,indx] <- K[,indx] + tmp   # add first "differences" columns
    }
  } else {
    if (args$differences != 0)
      stop(sQuote("differences"), " must be an non-neative integer")
    K <- diag(ncol(mm[[1]]))
  }

  if (vary != "" && ncol(by) > 1){       # build block diagonal penalty
    suppressMessages(K <- kronecker(diag(ncol(by)), K))
  }
  if (!identical(args$center, FALSE)) {
    tmp <- attributes(X)[c("degree", "knots", "Boundary.knots")]
    center <- match.arg(as.character(args$center),
                        choices = c("TRUE", "differenceMatrix", "spectralDecomp"))
    if (center == "TRUE") center <- "differenceMatrix"
    X <- switch(center,
                ### L = t(D) in Section 2.3. of Fahrmeir et al. (2004, Stat Sinica)
                "differenceMatrix" = tcrossprod(X, K) %*% solve(tcrossprod(K)),
                ### L = \Gamma \Omega^1/2 in Section 2.3. of
                ### Fahrmeir et al. (2004, Stat Sinica)
                "spectralDecomp" = {
                  SVD <- eigen(crossprod(K), symmetric = TRUE)
                  ev <- SVD$vector[, 1:(ncol(X) - args$differences), drop = FALSE]
                  ew <- SVD$values[1:(ncol(X) - args$differences), drop = FALSE]
                  X %*% ev %*% diag(1/sqrt(ew))
                }
    )
    attributes(X)[c("degree", "knots", "Boundary.knots")] <- tmp
    K <- diag(ncol(X))
```

```
  } else {
    K <- crossprod(K)
  }
  if (!is.null(attr(X, "Ts_constraint"))) {
    D <- attr(X, "D")
    K <- crossprod(D, K) %*% D
  }
}
if (length(mm) == 2) {
  suppressMessages(
    X <- kronecker(mm[[1]], matrix(1, ncol = ncol(mm[[2]]))) *
      kronecker(matrix(1, ncol = ncol(mm[[1]])), mm[[2]])
  )
  if (vary != "") {
    by <- model.matrix(as.formula(paste("~", vary, collapse = "")),
                       data = mf)[ , -1, drop = FALSE] # drop intercept
    DM <- X * by[,1]
    if (ncol(by) > 1){
      for (i in 2:ncol(by))
        DM <- cbind(DM, (X * by[,i]))
    }
    X <- DM
    ### <FIXME> Names of X if by is given
  }
  if (args$differences > 0){
    if (!args$cyclic) {
      Kx <- diff(diag(ncol(mm[[1]])), differences = args$differences)
      Ky <- diff(diag(ncol(mm[[2]])), differences = args$differences)
    } else {
      ## cyclic P-splines
      differences <- args$differences
      Kx <- diff(diag(ncol(mm[[1]]) + differences),
                 differences = differences)
      Ky <- diff(diag(ncol(mm[[2]]) + differences),
                 differences = differences)

      tmp <- Kx[,(1:differences)]   # save first "differences" columns
      Kx <- Kx[,-(1:differences)]    # drop first "differences" columns
      indx <- (ncol(mm[[1]]) - differences + 1):(ncol(mm[[1]]))
      Kx[,indx] <- Kx[,indx] + tmp   # add first "differences" columns

      tmp <- Ky[,(1:differences)]   # save first "differences" columns
```

```
      Ky <- Ky[,-(1:differences)]    # drop first "differences" columns
      indx <- (ncol(mm[[2]]) - differences + 1):(ncol(mm[[2]]))
      Ky[,indx] <- Ky[,indx] + tmp   # add first "differences" columns
    }
  } else {
    if (args$differences != 0)
      stop(sQuote("differences"), " must be an non-neative integer")
    Kx <- diag(ncol(mm[[1]]))
    Ky <- diag(ncol(mm[[2]]))
  }

  Kx <- crossprod(Kx)
  Ky <- crossprod(Ky)
  suppressMessages(
    K <- kronecker(Kx, diag(ncol(mm[[2]]))) +
      kronecker(diag(ncol(mm[[1]])), Ky)
  )
  if (vary != "" && ncol(by) > 1){       # build block diagonal penalty
    suppressMessages(K <- kronecker(diag(ncol(by)), K))
  }
  if (!identical(args$center, FALSE)) {
    ### L = \Gamma \Omega^1/2 in Section 2.3. of Fahrmeir et al.
    ### (2004, Stat Sinica), always
    L <- eigen(K, symmetric = TRUE)
    L$vectors <- L$vectors[,1:(ncol(X) - args$differences^2), drop = FALSE]
    L$values <- sqrt(L$values[1:(ncol(X) - args$differences^2), drop = FALSE])
    L <- L$vectors %*% (diag(length(L$values)) * (1/L$values))
    X <- as(X %*% L, "matrix")
    K <- as(diag(ncol(X)), "matrix")
  }
}
if (length(mm) > 2)
  stop("not possible to specify more than two variables in ",
       sQuote("..."), " argument of smooth base-learners")

## compare specified degrees of freedom to dimension of null space
if (!is.null(args$df)){
  rns <- ncol(K) - qr(as.matrix(K))$rank # compute rank of null space
  if (rns == args$df)
    warning( sQuote("df"), " equal to rank of null space ",
             "(unpenalized part of P-spline);\n  ",
             "Consider larger value for ", sQuote("df"),
             " or set ", sQuote("center != FALSE"), ".", immediate.=TRUE)
```

```
    if (rns > args$df)
      stop("not possible to specify ", sQuote("df"),
           " smaller than the rank of the null space\n  ",
           "(unpenalized part of P-spline). Use larger value for ",
           sQuote("df"), " or set ", sQuote("center != FALSE"), ".")
  }
  return(list(X = X, K = K))
}

### P-spline (and tensor-product spline) baselearner
bbs <- function(..., by = NULL, index = NULL, knots = 20, boundary.knots = NULL,
                degree = 3, differences = 2, df = 4, lambda = NULL, center = FALSE,
                cyclic = FALSE, constraint = c("none", "increasing", "decreasing"),
                deriv = 0) {

  if (!is.null(lambda)) df <- NULL

  cll <- match.call()
  cll[[1]] <- as.name("bbs")

  constraint <- match.arg(constraint)
  if (constraint != "none")
    warning("Using ", sQuote('bbs()'), ' with constraint != "none" is discouraged. Preferably use ',
            sQuote('bmono()'), " instead.\n",
            "See section ", sQuote("Details"), " of ?bbs for more information.")

  mf <- list(...)
  if (is.null(by)) {
    tmp <- mf
  } else {
    tmp <- c(mf, list(by))
  }
  if (length(unique(sapply(tmp, length))) > 1)
    warning("The elements in ... or by imply different number of rows: ",
            paste(unique(sapply(tmp, length)), collapse = ", "))
  rm("tmp")

  if (length(mf) == 1 && ((is.matrix(mf[[1]]) || is.data.frame(mf[[1]])) &&
                          ncol(mf[[1]]) > 1 )) {
    mf <- as.data.frame(mf[[1]])
  } else {
    mf <- as.data.frame(mf)
    cl <- as.list(match.call(expand.dots = FALSE))[2][[1]]
```

```
  colnames(mf) <- sapply(cl, function(x) deparse(x))
}
stopifnot(is.data.frame(mf))
if(!(all(sapply(mf, is.numeric)))) {
  if (ncol(mf) == 1){
    warning("cannot compute ", sQuote("bbs"),
            " for non-numeric variables; used ",
            sQuote("bols"), " instead.")
    return(bols(mf, by = by, index = index))
  }
  stop("cannot compute bbs for non-numeric variables")
}
vary <- ""
if (!is.null(by)){
  mf <- cbind(mf, by)
  colnames(mf)[ncol(mf)] <- vary <- deparse(substitute(by))
}

CC <- all(Complete.cases(mf))
if (!CC)
  warning("base-learner contains missing values;\n",
          "missing values are excluded per base-learner, ",
          "i.e., base-learners may depend on different",
          " numbers of observations.")
### option
DOINDEX <- (nrow(mf) > options("mboost_indexmin")[[1]])
if (is.null(index)) {
  if (!CC || DOINDEX) {
    index <- get_index(mf)
    mf <- mf[index[[1]],,drop = FALSE]
    index <- index[[2]]
  }
}

ret <- list(model.frame = function()
  if (is.null(index)) return(mf) else return(mf[index,,drop = FALSE]),
  get_call = function(){
    cll <- deparse(cll, width.cutoff=500L)
    if (length(cll) > 1)
      cll <- paste(cll, collapse="")
    cll
  },
```

```
    get_data = function() mf,
    get_index = function() index,
    get_vary = function() vary,
    get_names = function() colnames(mf),
    set_names = function(value) {
      if(length(value) != length(colnames(mf)))
        stop(sQuote("value"), " must have same length as ",
             sQuote("colnames(mf)"))
      for (i in 1:length(value)){
        cll[[i+1]] <<- as.name(value[i])
      }
      attr(mf, "names") <<- value
    })
  class(ret) <- "blg"

  ret$dpp <- bl_lin(ret, Xfun = X_bbs,
                    args = hyper_bbs(mf, vary, knots = knots, boundary.knots =
                                       boundary.knots, degree = degree, differences = differences,
                                     df = df, lambda = lambda, center = center, cyclic = cyclic,
                                     constraint = match.arg(constraint), deriv = deriv))
  return(ret)
}

### cyclic B-splines
### adapted version of mgcv::cSplineDes from S.N. Wood
cbs <- function (x, knots, boundary.knots, degree = 3, deriv = 0L) {

  if (any(x < boundary.knots[1], na.rm = TRUE) |
      any(x > boundary.knots[2], na.rm = TRUE))
    stop("some ", sQuote("x"), " values are beyond ",
         sQuote("boundary.knots"))

  nx <- names(x)
  x <- as.vector(x)
  ## handling of NAs
  nax <- is.na(x)
  if (nas <- any(nax))
    x <- x[!nax]

  knots <- c(boundary.knots[1], knots, boundary.knots[2])
  nKnots <- length(knots)
  ord <- degree + 1
  xc <- knots[nKnots - ord + 1]
```

```
  knots <- c(boundary.knots[1] -
               (boundary.knots[2] - knots[(nKnots - ord + 1):(nKnots - 1)]),
             knots)
  ind <- x > xc
  X <- splineDesign(knots, x, ord, derivs = rep(deriv, length(x)), outer.ok = TRUE)
  x[ind] <- x[ind] - boundary.knots[2] + boundary.knots[1]
  if (sum(ind)) {
    Xtmp <- splineDesign(knots, x[ind], ord, derivs = rep(deriv, length(x[ind])),
                         outer.ok = TRUE)
    X[ind, ] <- X[ind, ] + Xtmp
  }
  ## handling of NAs
  if (nas) {
    tmp <- matrix(NA, length(nax), ncol(X))
    tmp[!nax, ] <- X
    X <- tmp
  }
  ## add attributes
  attr(X, "degree") <- degree
  attr(X,"knots") <- knots
  attr(X,"boundary.knots") <- boundary.knots
  if (length(deriv) > 1 || deriv != 0)
    attr(X, "deriv") <- deriv
  dimnames(X) <- list(nx, 1L:ncol(X))
  return(X)
}


bsplines <- function(x, knots, boundary.knots, degree,
                     Ts_constraint = "none", deriv = 0L,
                     extrapolation = FALSE) {


  ## do not allow data beyond boundary knots while fitting
  if (!extrapolation && (any(x < boundary.knots[1], na.rm = TRUE) |
                         any(x > boundary.knots[2], na.rm = TRUE)))
    stop("some ", sQuote("x"), " values are beyond ",
         sQuote("boundary.knots"))


  ## allow extrapolation when predicting
  if (extrapolation <- extrapolation &&
      (any(x < boundary.knots[1], na.rm = TRUE) |
       any(x > boundary.knots[2], na.rm = TRUE))) {
    warning("Some ", sQuote("x"), " values are beyond ",
```

```
          sQuote("boundary.knots"), "; Linear extrapolation used.")
}

nx <- names(x)
x <- as.vector(x)
## handling of NAs
nax <- is.na(x)
if (nas <- any(nax))
  x <- x[!nax]
## use equidistant boundary knots
dx <- diff(boundary.knots)/(length(knots) + 1)
bk_lower <- seq(boundary.knots[1] - degree * dx, boundary.knots[1],
                length = degree + 1)
bk_upper <- seq(boundary.knots[2], boundary.knots[2] + degree * dx,
                length = degree + 1)
## complete knot mesh
k <- c(bk_lower, knots, bk_upper)
## construct design matrix
X <- splineDesign(k, x, degree + 1, derivs = rep(deriv, length(x)),
                  outer.ok = TRUE)

## code along the lines of mgcv::Predict.matrix.pspline.smooth
if (extrapolation) {
  ## Build matrix to map coeficients to value (deriv = 0) and
  ## slope (deriv = 1) at end points.
  if (deriv != 0L) {
    warning("deriv != 0L; Linear extrapolation overwritten")
  } else {
    deriv <- c(0, 1, 0, 1)
  }
  D <- splineDesign(knots = k, x = rep(boundary.knots, each = 2),
                    ord = degree + 1, deriv)
  ## Add rows for linear extrapolation
  idx <- x < boundary.knots[1]
  if (any(idx, na.rm = TRUE))
    X[idx,] <- cbind(1, x[idx] - boundary.knots[1]) %*% D[1:2, ]
  idx <- x > boundary.knots[2]
  if (any(idx, na.rm = TRUE))
    X[idx,] <- cbind(1, x[idx] - boundary.knots[2]) %*% D[3:4, ]
}

## handling of NAs
if (nas) {
```

```
    tmp <- matrix(NA, length(nax), ncol(X))
    tmp[!nax, ] <- X
    X <- tmp
  }
  ### constraints; experimental
  D <- diag(ncol(X))
  D[lower.tri(D)] <- 1
  X <- switch(Ts_constraint, "none" = X,
              "increasing" = X %*% D,
              "decreasing" = -X %*% D)
  ## add attributes
  attr(X, "degree") <- degree
  attr(X, "knots") <- knots
  attr(X, "boundary.knots") <- list(lower = bk_lower, upper = bk_upper)
  if (Ts_constraint != "none")
    attr(X, "Ts_constraint") <- Ts_constraint
  if (Ts_constraint != "none")
    attr(X, "D") <- D
  if (length(deriv) > 1 || deriv != 0)
    attr(X, "deriv") <- deriv
  dimnames(X) <- list(nx, 1L:ncol(X))
  return(X)
}
```