```
### copyright Dormann, C and Kaschner, K, additional material to MCED ###
#  when using, or when questions, please e-mail carsten.dormann@ufz.de, #
###             kristin.kaschner@biologie.uni-freiburg.de             ###

## some additional code to facilitate the analysis presented in Dormann & Kaschner (2011)

require(gbm)
require(verification)
require(ncf)
require(randomForest)
require(Hmisc)
require(MASS)
require(maptools)
require(spdep)

formula.maker <- function(dataframe, y.col=1, quadratic=TRUE, interactions=TRUE){
  # makes a formula for GLM from dataframe column names,
  # including quadratic effects and first-order interactions
  # by default, first column is taken to be the response (y); else, an integer giving the column with
the response in "dataframe"
  # by Carsten F. Dormann
  if (quadratic && interactions) {
      f <- as.formula(paste(colnames(dataframe)[y.col], " ~ (", paste(colnames(dataframe[,-y.col]),
collapse=" + ", sep=""), ")^2 + ", paste("I(", colnames(dataframe[,-y.col]), "^2)", collapse="+",
sep="")))
  }

  if (quadratic & !interactions){
      f <- as.formula(paste(colnames(dataframe)[y.col], " ~ (", paste(colnames(dataframe[,-y.col]),
collapse=" + ", sep=""), ") + ", paste("I(", colnames(dataframe[,-y.col]), "^2)", collapse="+",
sep="")))
  }

  if (!quadratic & !interactions){
      f <- as.formula(paste(colnames(dataframe)[y.col], " ~ ", paste(colnames(dataframe[,-y.col]),
collapse=" + ", sep="") ))
  }

  if (!quadratic & interactions){
      f <- as.formula(paste(colnames(dataframe)[y.col], " ~ (", paste(colnames(dataframe[,-y.col]),
collapse=" + ", sep=""), ")^2"))
      # + ", paste("I(", colnames(dataframe[,-1]), "^2)", collapse="+", sep="")))
  }

  f
}

## here the original file starts


`gbm.step` <-
function (data,                                 # the input dataframe
  gbm.x,                                        # the predictors
  gbm.y,                                        # and response
  tree.complexity = 1,                          # sets the complexity of individual trees
  learning.rate = 0.001,                        # sets the weight applied to inidivudal trees
  bag.fraction = 0.5,                           # sets the proportion of observations used in selecting
variables
  site.weights = rep(1, nrow(data)),            # allows varying weighting for sites
  var.monotone = rep(0, length(gbm.x)),         # restricts responses to individual predictors to monotone
  n.folds = 10,                                 # number of folds
  prev.stratify = TRUE,                         # prevalence stratify the folds - only for p/a data
  family = "bernoulli",                         # family - bernoulli (=binomial), poisson, laplace or
gaussian
  n.trees = 50,                                 # number of initial trees to fit
  step.size = n.trees,                          # numbers of trees to add at each cycle
  max.trees = 5000,                             # max number of trees to fit before stopping
  tolerance.method = "fixed",                   # method to use in deciding to stop - "fixed" or "auto"
  tolerance = 0.0,                              # tolerance value to use if method == fixed
  keep.data = FALSE,                            # keep raw data in final model
  plot.main = TRUE,                             # plot hold-out deviance curve
  plot.folds = FALSE,                           # plot the individual folds as well
```

```
  verbose = TRUE,                          # control amount of screen reporting
  silent = FALSE,                          # to allow running with no output for simplifying model)
  ...)                                     # allows for any additional plotting parameters
{
#
# j. leathwick/j. elith - 30th March 2007
#
# version 2.8
#
# function to assess optimal no of boosting trees using k-fold cross validation
#
# implements the cross-validation procedure described on page 215 of
# Hastie T, Tibshirani R, Friedman JH (2001) The Elements of Statistical Learning:
# Data Mining, Inference, and Prediction Springer-Verlag, New York.
#
# divides the data into 10 subsets, with stratification by prevalence if required for pa data
# then fits a gbm model of increasing complexity along the sequence from n.trees to n.trees + (n.steps
* step.size)
# calculating the residual deviance at each step along the way
# after each fold processed, calculates the average holdout residual deviance and its standard error
# then identifies the optimal number of trees as that at which the holdout deviance is minimised
# and fits a model with this number of trees, returning it as a gbm model along with additional
information
# from the cv selection process
#
# updated 13/6/05 to accommodate weighting of sites
#
# updated 19/8/05 to increment all folds simultaneously, allowing the stopping rule
# for the maxinum number of trees to be fitted to be imposed by the data,
# rather than being fixed in advance
#
# updated 29/8/05 to return cv test statistics, and deviance as mean
# time for analysis also returned via unclass(Sys.time())
#
# updated 5/9/05 to use external function calc.deviance
# and to return cv test stats via predictions formed from fold models
# with n.trees = target.trees
#
# updated 15/5/06 to calculate variance of fitted and predicted values across folds
# these can be expected to approximate the variance of fitted values
# as would be estimated for example by bootstrapping
# as these will underestimate the true variance
# they are corrected by multiplying by (n-1)2/n
# where n is the number of folds
#
# updated 25/3/07 to allow varying of bag fraction
#
# requires gbm library from Cran
# requires roc and calibration scripts of J Elith
# requires calc.deviance script of J Elith/J Leathwick
#
#

  require(gbm)

  if (silent) verbose <- FALSE

# initiate timing call

  z1 <- unclass(Sys.time())

# setup input data and assign to position one

  dataframe.name <- deparse(substitute(data))   # get the dataframe name

  data <- eval(data)
  x.data <- eval(data[, gbm.x])                 #form the temporary datasets
  names(x.data) <- names(data)[gbm.x]
  y.data <- eval(data[, gbm.y])
  sp.name <- names(data)[gbm.y]

  assign("x.data", x.data, env = globalenv())               #and assign them for later use
```

```
  assign("y.data", y.data, env = globalenv())

  n.cases <- nrow(data)

  if (!silent) {
    cat("\n","\n","GBM STEP - version 2.5","\n","\n")
    cat("Performing cross-validation selection of a boosted regression tree model for",sp.name,"\n")
    cat(" with dataframe",dataframe.name,"containing",n.cases,"rows of data","\n","\n")
  }

# set up the selector variable either with or without prevalence stratification

  if (prev.stratify & family == "bernoulli") {
    presence.mask <- data[,gbm.y] == 1
    absence.mask <- data[,gbm.y] == 0
    n.pres <- sum(presence.mask)
    n.abs <- sum(absence.mask)

# create a vector of randomised numbers and feed into presences
    selector <- rep(0,n.cases)
    temp <- rep(seq(1, n.folds, by = 1), length = n.pres)
    temp <- temp[order(runif(n.pres, 1, 100))]
    selector[presence.mask] <- temp

# and then do the same for absences
    temp <- rep(seq(1, n.folds, by = 1), length = n.abs)
    temp <- temp[order(runif(n.abs, 1, 100))]
    selector[absence.mask] <- temp
    }

  else {  #otherwise make them random with respect to presence/absence
    selector <- rep(seq(1, n.folds, by = 1), length = n.cases)
    selector <- selector[order(runif(n.cases, 1, 100))]
    }

# set up the storage space for results

  pred.values <- rep(0, n.cases)

  cv.loss.matrix <- matrix(0, nrow = n.folds, ncol = 1)
  training.loss.matrix <- matrix(0, nrow = n.folds, ncol = 1)

  trees.fitted <- n.trees

  model.list <- list(paste("model",c(1:n.folds),sep=""))     # dummy list for the tree models

# set up the initial call to gbm

  gbm.call <- paste("gbm(y.subset ~ .,data=x.subset, n.trees = n.trees,
    interaction.depth = tree.complexity, shrinkage = learning.rate,
    bag.fraction = bag.fraction, weights = weight.subset,
    distribution = as.character(family), var.monotone = var.monotone,
    verbose = FALSE)", sep="")

  n.fitted <- n.trees

# calculate the total deviance

  y_i <- y.data

  u_i <- sum(y.data * site.weights) / sum(site.weights)
  u_i <- rep(u_i,length(y_i))

  total.deviance <- calc.deviance(y_i, u_i, weights = site.weights, family = family, calc.mean = FALSE)

  mean.total.deviance <- total.deviance/n.cases

# now step through the folds setting up the initial call

  if (!silent){

    cat("creating",n.folds,"initial models of",n.trees,"trees","\n")
```

```
    if (prev.stratify & family == "bernoulli") cat("\n","folds are stratified by prevalence","\n","\n")
      else cat("\n","folds are unstratified","\n","\n")

    if (tolerance.method == "fixed") {
      cat("tolerance is fixed at ",tolerance,"\n","\n")}
    if(tolerance.method == "auto") {
      cat("tolerance will be fixed at 1% of the first change in deviance","\n","\n")}
    if (tolerance.method != "fixed" & tolerance.method != "auto") {
      cat("invalid argument for tolerance method - should be auto or fixed","\n")
      return()}

    cat ("total mean deviance = ",round(mean.total.deviance,4),"\n","\n")
  }

  if (verbose) cat("ntrees resid. dev.","\n")

  for (i in 1:n.folds) {

    model.mask <- selector != i  #used to fit model on majority of data
    pred.mask <- selector == i   #used to identify the with-held subset

    y.subset <- y.data[model.mask]
    x.subset <- x.data[model.mask,]
    weight.subset <- site.weights[model.mask]

    model.list[[i]] <- eval(parse(text = gbm.call))

    fitted.values <- predict.gbm(model.list[[i]], x.subset, type = "response", n.trees = n.trees)
    pred.values[pred.mask] <- predict.gbm(model.list[[i]], x.data[pred.mask, ], type = "response",
n.trees = n.trees)

# calc training deviance

    y_i <- y.subset
    u_i <- fitted.values
    weight.fitted <- site.weights[model.mask]
    training.loss.matrix[i,1] <- calc.deviance(y_i, u_i, weight.fitted, family = family)

# calc holdout deviance

    y_i <- y.data[pred.mask]
    u_i <- pred.values[pred.mask]
    weight.preds <- site.weights[pred.mask]
    cv.loss.matrix[i,1] <- calc.deviance(y_i, u_i, weight.preds, family = family)

  } # end of first loop

# now process until the change in mean deviance is =< tolerance or max.trees is exceeded

  delta.deviance <- 1

  cv.loss.values <- apply(cv.loss.matrix,2,mean)
  if (verbose) cat(n.fitted,"  ",round(cv.loss.values,4),"\n","\n")

  if (!silent) cat("")
  if (!silent) cat("now adding trees...","\n")

  j <- 1

  while (delta.deviance > tolerance & n.fitted < max.trees) {  # beginning of inner loop

# add a new column to the results matrice..

    training.loss.matrix <- cbind(training.loss.matrix,rep(0,n.folds))
    cv.loss.matrix <- cbind(cv.loss.matrix,rep(0,n.folds))

    n.fitted <- n.fitted + step.size
    trees.fitted <- c(trees.fitted,n.fitted)

    j <- j + 1
```

```
    for (i in 1:n.folds) {

      model.mask <- selector != i  #used to fit model on majority of data
      pred.mask <- selector == i   #used to identify the with-held subset

      y.subset <- y.data[model.mask]
      x.subset <- x.data[model.mask,]
      weight.subset <- site.weights[model.mask]

      model.list[[i]] <- gbm.more(model.list[[i]], step.size)

      fitted.values <- predict.gbm(model.list[[i]],x.subset, type = "response", n.trees = n.fitted)
      pred.values[pred.mask] <- predict.gbm(model.list[[i]], x.data[pred.mask, ], type = "response",
n.trees = n.fitted)

# calculate training deviance

      y_i <- y.subset
      u_i <- fitted.values
      weight.fitted <- site.weights[model.mask]
      training.loss.matrix[i,j] <- calc.deviance(y_i, u_i, weight.fitted, family = family)

# calc holdout deviance

      u_i <- pred.values[pred.mask]
      y_i <- y.data[pred.mask]
      weight.preds <- site.weights[pred.mask]
      cv.loss.matrix[i,j] <- calc.deviance(y_i, u_i, weight.preds, family = family)

    }  # end of inner loop

    cv.loss.values <- apply(cv.loss.matrix,2,mean)

    if (j == 2) {
      if (tolerance.method == "auto") {
        tolerance <- (cv.loss.values[1] - cv.loss.values[2]) / 1000
        if (!silent) cat("tolerance set to ",round(tolerance,5),"\n")}
      }

      if (j < 5) {
        if (cv.loss.values[j] > cv.loss.values[j-1]) {
          if (!silent) cat("restart model with a smaller learning rate or smaller step size...")
          return()}
           }

      if (j >= 20) {   #calculate stopping rule value
        test1 <- mean(cv.loss.values[(j-9):j])
        test2 <- mean(cv.loss.values[(j-19):(j-9)])
        delta.deviance <- test2 - test1
      }

      if (verbose) cat(n.fitted," ",round(cv.loss.values[j],4),"\n")

  } # end of while loop

# now begin process of calculating optimal number of trees

  training.loss.values <- apply(training.loss.matrix,2,mean)

  cv.loss.ses <- rep(0,length(cv.loss.values))
  cv.loss.ses <- sqrt(apply(cv.loss.matrix,2,var)) / sqrt(n.folds)

# find the target holdout deviance

  y.bar <- min(cv.loss.values)

# plot out the resulting curve of holdout deviance

  if (plot.main) {

    y.min <- min(cv.loss.values - cv.loss.ses)  #je added multiplier 10/8/05
    y.max <- max(cv.loss.values + cv.loss.ses)  #je added multiplier 10/8/05 }
```

```
    if (plot.folds) {
      y.min <- min(cv.loss.matrix)
      y.max <- max(cv.loss.matrix) }

      plot(trees.fitted, cv.loss.values, type = 'l', ylab = "holdout deviance",
          xlab = "no. of trees", ylim = c(y.min,y.max), ...)
      abline(h = y.bar, col = 2)

      lines(trees.fitted, cv.loss.values + cv.loss.ses, lty=2)
      lines(trees.fitted, cv.loss.values - cv.loss.ses, lty=2)

      if (plot.folds) {
        for (i in 1:n.folds) {
          lines(trees.fitted, cv.loss.matrix[i,],lty = 3)
      }
    }
  }

# identify the optimal number of trees

  target.trees <- trees.fitted[match(TRUE,cv.loss.values == y.bar)]

  if(plot.main) {
    abline(v = target.trees, col=3)
    title(paste(sp.name,", d - ",tree.complexity,", lr - ",learning.rate, sep=""))
  }

# estimate the cv deviance and test statistics
# includes estimates of the standard error of the fitted values added 2nd may 2005

  cv.deviance.stats <- rep(0, n.folds)
  cv.roc.stats <- rep(0, n.folds)
  cv.cor.stats <- rep(0, n.folds)
  cv.calibration.stats <- matrix(0, ncol=5, nrow = n.folds)

  fitted.matrix <- matrix(NA, nrow = n.cases, ncol = n.folds)  # used to calculate se's

  for (i in 1:n.folds) {

    pred.mask <- selector == i   #used to identify the with-held subset
    model.mask <- selector != i  #used to fit model on majority of data

    fitted.matrix[model.mask,i] <- predict.gbm(model.list[[i]], x.data[model.mask, ], type =
"response", n.trees = target.trees)
    fitted.matrix[pred.mask,i]  <- predict.gbm(model.list[[i]], x.data[pred.mask, ], type =
"response", n.trees = target.trees)

    y_i <- y.data[pred.mask]
    u_i <- fitted.matrix[pred.mask,i]  #pred.values[pred.mask]
    weight.preds <- site.weights[pred.mask]

    cv.deviance.stats[i] <- calc.deviance(y_i, u_i, weight.preds, family = family)

    cv.cor.stats[i] <- cor(y_i,u_i)

    if (family == "bernoulli") {
      cv.roc.stats[i] <- roc(y_i,u_i)
      cv.calibration.stats[i,] <- calibration(y_i,u_i,"binomial")
    }

    if (family == "poisson") {
      cv.calibration.stats[i,] <- calibration(y_i,u_i,"poisson")
    }
  }

  fitted.vars <- apply(fitted.matrix,1, var, na.rm = TRUE)

# now calculate the mean and se's for the folds

  cv.dev <- mean(cv.deviance.stats, na.rm = TRUE)
  cv.dev.se <- sqrt(var(cv.deviance.stats)) / sqrt(n.folds)
```

```
  cv.cor <- mean(cv.cor.stats, na.rm = TRUE)
  cv.cor.se <- sqrt(var(cv.cor.stats, use = "complete.obs")) / sqrt(n.folds)

  cv.roc <- 0.0
  cv.roc.se <- 0.0

  if (family == "bernoulli") {
    cv.roc <- mean(cv.roc.stats,na.rm=TRUE)
    cv.roc.se <- sqrt(var(cv.roc.stats, use = "complete.obs")) / sqrt(n.folds)  }

  cv.calibration <- 0.0
  cv.calibration.se <- 0.0

  if (family == "poisson" | family == "bernoulli") {
    cv.calibration <- apply(cv.calibration.stats,2,mean)
    cv.calibration.se <- apply(cv.calibration.stats,2,var)
    cv.calibration.se <- sqrt(cv.calibration.se) / sqrt(n.folds) }

# fit the final model

  gbm.call <- paste("gbm(y.data ~ .,n.trees = target.trees,
    data = x.data, verbose = F, interaction.depth = tree.complexity,
    bag.fraction = bag.fraction, weights = site.weights,
    shrinkage = learning.rate, distribution = as.character(family),
    var.monotone = var.monotone, keep.data = keep.data)", sep="")

  if (!silent) cat("fitting final gbm model with a fixed number of ",target.trees," trees for
",sp.name,"\n")

  gbm.object <- eval(parse(text = gbm.call))

  best.trees <- target.trees

# extract fitted values and summary table

  gbm.summary <- summary(gbm.object,n.trees = target.trees, plotit = FALSE)

  fitted.values <- predict.gbm(gbm.object,x.data,n.trees = target.trees,type="response")

  y_i <- y.data
  u_i <- fitted.values
  resid.deviance <- calc.deviance(y_i, u_i, weights = site.weights, family = family, calc.mean = FALSE)

  self.cor <- cor(y_i,u_i)
  self.calibration <- 0.0
  self.roc <- 0.0

  if (family == "bernoulli") {  # do this manually as we need the residuals
    deviance.contribs <- (y_i * log(u_i)) + ((1-y_i) * log(1 - u_i))
    residuals <- sqrt(abs(deviance.contribs * 2))
    residuals <- ifelse((y_i - u_i) < 0, 0 - residuals, residuals)
    self.roc <- roc(y_i,u_i)
    self.calibration <- calibration(y_i,u_i,"binomial")
  }

  if (family == "poisson") {   # do this manually as we need the residuals
    deviance.contribs <- ifelse(y_i == 0, 0, (y_i * log(y_i/u_i))) - (y_i - u_i)
    residuals <- sqrt(abs(deviance.contribs * 2))
    residuals <- ifelse((y_i - u_i) < 0, 0 - residuals, residuals)
    self.calibration <- calibration(y_i,u_i,"poisson")
  }

  if (family == "gaussian" | family == "laplace") {
    residuals <- y_i - u_i
  }

  mean.resid.deviance <- resid.deviance/n.cases

  z2 <- unclass(Sys.time())
  elapsed.time.minutes <- round((z2 - z1)/ 60,2)  #calculate the total elapsed time
```

```
  if (verbose) {
    cat("\n")
    cat("mean total deviance =", round(mean.total.deviance,3),"\n")
    cat("mean residual deviance =", round(mean.resid.deviance,3),"\n","\n")
    cat("estimated cv deviance =", round(cv.dev,3),"; se =",
      round(cv.dev.se,3),"\n","\n")
    cat("training data correlation =",round(self.cor,3),"\n")
    cat("cv correlation = ",round(cv.cor,3),"; se =",round(cv.cor.se,3),"\n","\n")
    if (family == "bernoulli") {
      cat("training data ROC score =",round(self.roc,3),"\n")
      cat("cv ROC score =",round(cv.roc,3),"; se =",round(cv.roc.se,3),"\n","\n")
    }
    cat("elapsed time - ",round(elapsed.time.minutes,2),"minutes","\n")
  }

  if (n.fitted == max.trees & !silent) {
    cat("\n","########### warning ##########","\n","\n")
    cat("maximum tree limit reached - results may not be optimal","\n")
    cat("  - refit with faster learning rate or increase maximum number of trees","\n")
  }

# now assemble data to be returned

  gbm.detail <- list(dataframe = dataframe.name, gbm.x = gbm.x, predictor.names = names(x.data),
    gbm.y = gbm.y, response.name = sp.name, family = family, tree.complexity = tree.complexity,
    learning.rate = learning.rate, bag.fraction = bag.fraction, cv.folds = n.folds,
    prev.stratification = prev.stratify, max.fitted = n.fitted, n.trees = target.trees,
    best.trees = target.trees, train.fraction = 1.0, tolerance.method = tolerance.method,
    tolerance = tolerance, var.monotone = var.monotone, date = date(), ### random.seed = seed,
    elapsed.time.minutes = elapsed.time.minutes)

  training.stats <- list(null = total.deviance, mean.null = mean.total.deviance,
    resid = resid.deviance, mean.resid = mean.resid.deviance, correlation = self.cor,
    discrimination = self.roc, calibration = self.calibration)

  cv.stats <- list(deviance.mean = cv.dev, deviance.se = cv.dev.se,
    correlation.mean = cv.cor, correlation.se = cv.cor.se,
    discrimination.mean = cv.roc, discrimination.se = cv.roc.se,
    calibration.mean = cv.calibration, calibration.se = cv.calibration.se)

  rm(x.data,y.data, envir = globalenv())           #finally, clean up the temporary dataframes

  for (i in 1:n.folds) model.list[[i]] <- NULL
  x.data <- NULL
  y.data <- NULL
  pred.values <- NULL

# and assemble results for return

  gbm.object$gbm.call <- gbm.detail
  gbm.object$fitted <- fitted.values
  gbm.object$fitted.vars <- fitted.vars
  gbm.object$residuals <- residuals
  gbm.object$contributions <- gbm.summary
  gbm.object$self.statistics <- training.stats
  gbm.object$cv.statistics <- cv.stats
  gbm.object$weights <- site.weights
  gbm.object$trees.fitted <- trees.fitted
  gbm.object$training.loss.values <- training.loss.values
  gbm.object$cv.values <- cv.loss.values
  gbm.object$cv.loss.ses <- cv.loss.ses
  gbm.object$cv.loss.matrix <- cv.loss.matrix
  gbm.object$cv.roc.matrix <- cv.roc.stats

  return(gbm.object)
}

`gbm.fixed` <-
function (data,
  gbm.x,
  gbm.y,
  tree.complexity = 1,
```

```
  site.weights = rep(1, nrow(data)),
  verbose = TRUE,
  learning.rate = 0.001,
  n.trees = 2000,
  train.fraction = 1,
  family = "bernoulli",
  keep.data = FALSE,
  var.monotone = rep(0, length(gbm.x))
  )
{
#
# j leathwick, j elith - 6th May 2006
#
# version 2.5 - developed in R 2.0
#
# calculates a gradient boosting (gbm)object with a fixed number of trees
# with the number of trees identified using gbm.step or some other procedure
#
# takes as input a dataset and args selecting x and y variables, and degree of interaction depth
#
# updated 13/6/05 to accommodate weighting of sites when calculating total and residual deviance
#
# updated 10/8/05 to correct how site.weights are returned
#
# requires gbm
#
#
    require(gbm)

# setup input data and assign to position one

    dataframe.name <- deparse(substitute(data))   # get the dataframe name

    x.data <- eval(data[, gbm.x])                 #form the temporary datasets
    names(x.data) <- names(data)[gbm.x]
    y.data <- eval(data[, gbm.y])
    sp.name <- names(data)[gbm.y]


    assign("x.data", x.data, pos = 1)             #and assign them for later use
    assign("y.data", y.data, pos = 1)

# fit the gbm model

    gbm.call <- paste("gbm(y.data ~ .,n.trees = n.trees, data=x.data, verbose = F, interaction.depth =
tree.complexity,
           weights = site.weights, shrinkage = learning.rate, distribution = as.character(family),
           var.monotone = var.monotone, keep.data = keep.data)", sep="")

    if (verbose) {
        print(paste("fitting gbm model with a fixed number of ",n.trees," trees for
",sp.name,sep=""),quote=FALSE) }

    gbm.object <- eval(parse(text = gbm.call))

    best.trees <- n.trees

#extract fitted values and summary table

    fitted.values <- predict.gbm(gbm.object,x.data,n.trees = n.trees,type="response")
    gbm.summary <- summary(gbm.object,n.trees = n.trees, plotit = FALSE)

    y_i <- y.data
    u_i <- fitted.values

    if (family == "poisson") {
        deviance.contribs <- ifelse(y_i == 0, 0, (y_i * log(y_i/u_i))) - (y_i - u_i)
        resid.deviance <- 2 * sum(deviance.contribs * site.weights)
        residuals <- sqrt(abs(deviance.contribs * 2))
        residuals <- ifelse((y_i - u_i) < 0, 0 - residuals, residuals)

        u_i <- sum(y.data * site.weights) / sum(site.weights)
```

```
        deviance.contribs <- ifelse(y_i == 0, 0, (y_i * log(y_i/u_i))) - (y_i - u_i)
        total.deviance <- 2 * sum(deviance.contribs * site.weights)
    }

    if (family == "bernoulli") {
        deviance.contribs <- (y_i * log(u_i)) + ((1-y_i) * log(1 - u_i))
        resid.deviance <- -2 * sum(deviance.contribs * site.weights)
        residuals <- sqrt(abs(deviance.contribs * 2))
        residuals <- ifelse((y_i - u_i) < 0, 0 - residuals, residuals)

        u_i <- sum(y.data * site.weights) / sum(site.weights)
        deviance.contribs <- (y_i * log(u_i)) + ((1-y_i) * log(1 - u_i))
        total.deviance <- -2 * sum(deviance.contribs * site.weights)
    }

    if (family == "laplace") {
        resid.deviance <- sum(abs(y_i - u_i))
        residuals <- y_i - u_i
        u_i <- mean(y.data)
        total.deviance <- sum(abs(y_i - u_i))
    }

    if (family == "gaussian") {
        resid.deviance <- sum((y_i - u_i) * (y_i - u_i))
        residuals <- y_i - u_i
        u_i <- mean(y.data)
        total.deviance <- sum((y_i - u_i) * (y_i - u_i))
    }

    if (verbose) {
        print(paste("total deviance = ",round(total.deviance,2),sep=""),quote=F)
        print(paste("residual deviance = ",round(resid.deviance,2),sep=""),quote=F)}

# now assemble data to be returned

    gbm.detail <- list(dataframe = dataframe.name, gbm.x = gbm.x, predictor.names = names(x.data),
        gbm.y = gbm.y, reponse.name = names(y.data), tree.complexity = tree.complexity, n.trees =
n.trees,
        learning.rate = learning.rate, best.trees = best.trees, cv.folds = 0,
        family = family, train.fraction = train.fraction, var.monotone = var.monotone)

    gbm.object$gbm.call <- gbm.detail
    gbm.object$fitted <- fitted.values
    gbm.object$residuals <- residuals
    gbm.object$contributions <- gbm.summary
    gbm.object$self.statistics <- list(null.deviance = total.deviance, resid.deviance = resid.deviance)
    gbm.object$weights <- site.weights

    rm(x.data,y.data, pos=1)           #finally, clean up the temporary dataframes

    return(gbm.object)
}

`gbm.holdout` <-
function (data,                        # the input data frame
   gbm.x,                              # indices of predictor variables
   gbm.y,                              # index of response variable
   learning.rate = 0.001,              # typically varied between 0.1 and 0.001
   tree.complexity = 1,                # sometimes called interaction depth
   family = "bernoulli",               # "bernoulli","poisson", etc. as for gbm
   n.trees = 200,                      # initial number of trees
   add.trees = n.trees,                # number of trees to add at each increment
   max.trees = 20000,                  # maximum number of trees to fit
   verbose = TRUE,                     # controls degree of screen reporting
   train.fraction = 0.8,               # proportion of data to use for training
   permute = TRUE,                     # reorder data to start with
   prev.stratify = TRUE,               # stratify selection for p/a data
   var.monotone = rep(0, length(gbm.x)),# allows constraining of response to monotone
   site.weights = rep(1, nrow(data)),  # set equal to 1 by default
   refit = TRUE,                       # refit the model with the full data but id'd no of trees
   keep.data = TRUE)                   # keep copy of the data
{
```

```
#
# j leathwick, j elith - October 2006
#
# version 2.7 - developed in R 2.3.1
#
# calculates a gradient boosting (gbm)object in which model complexity is
# determined using a training set with predictions made to a withheld set
# an initial set of trees is fitted, and then trees are progressively added
# testing performance # along the way, using gbm.perf until the optimal
# number of trees is identified
#
# as any structured ordering of the data should be avoided, a copy of the data set
# BY DEFAULT is randomly reordered each time the function is run
#
# takes as input a dataset and args selecting x and y variables, and degree of interaction depth
#
# requires gbm
#
#
  require(gbm)

# setup input data and assign to position one

  dataframe.name <- deparse(substitute(data))  # get the dataframe name
  cv.folds <- 0

  if (permute) {
    print("",quote=FALSE)
    print("WARNING - data is being randomly reordered to avoid confounding effects",quote=FALSE)
    print("of inherent structure as submitted - use permute = FALSE to turn off this
option",quote=FALSE)
    n.rows <- nrow(data)

    if (prev.stratify == TRUE & family == "bernoulli") {

      presence.mask <- data[,gbm.y] == 1
      absence.mask <- data[,gbm.y] == 0
      n.pres <- sum(presence.mask)
      n.abs <- sum(absence.mask)

      selector <- seq(1,n.rows)

      temp <- sample(selector[presence.mask],size = n.pres * train.fraction)
      selector[temp] <- 0

      temp <- sample(selector[absence.mask],size = n.abs * train.fraction)
      selector[temp] <- 0

      sort.vector <- sort(selector,index.return = TRUE)[[2]]
    }

    else {
      sort.vector <- sample(seq(1,n.rows),n.rows,replace=FALSE)
    }

    sort.data <- data[sort.vector,]

    x.data <- eval(sort.data[, gbm.x])                #form the temporary datasets
    y.data <- eval(sort.data[, gbm.y])
  }
  else {
    x.data <- eval(data[, gbm.x])                  #form the temporary datasets
    y.data <- eval(data[, gbm.y])
  }

  names(x.data) <- names(data)[gbm.x]
  sp.name <- names(data)[gbm.y]

  assign("x.data", x.data, pos = 1)                #and assign them for later use
  assign("y.data", y.data, pos = 1)

# fit the gbm model
```

```
  print(paste("fitting initial gbm model of ",n.trees," trees for ",sp.name,sep=""),quote=FALSE)
  print(" and expanding using withheld data for evaluation",quote=FALSE)

  gbm.call <- paste("gbm(y.data ~ .,n.trees = n.trees, data=x.data, verbose = F, interaction.depth =
tree.complexity,
    weights = site.weights, shrinkage = learning.rate, cv.folds = 0, distribution = as.character
(family),
    train.fraction = train.fraction, var.monotone = var.monotone, keep.data = keep.data)", sep="")

  gbm.object <- eval(parse(text = gbm.call))

# identify the best number of trees using method appropriate to model

  best.trees <- gbm.perf(gbm.object, method = 'test', plot.it = FALSE)

  n.fitted <- n.trees

  if (verbose) print("expanding model to find optimal no of trees...",quote=FALSE)

  while(gbm.object$n.trees - best.trees < n.trees & n.fitted < max.trees){

    gbm.object <- gbm.more(gbm.object, add.trees)
    best.trees <- gbm.perf(gbm.object, method = 'test', plot.it = FALSE)
    n.fitted <- n.fitted + add.trees

    if (n.fitted %% 100 == 0){ #report times along the way
      if (verbose) print(paste("fitted trees = ", n.fitted, sep = ""), quote = FALSE)
    }
  }

  if (verbose) print(paste("fitting stopped at ",best.trees," trees",sep=""),quote=FALSE)

  if (refit) {  # we are refitting the model with fixed tree size
    print(paste("refitting the model to the full dataset using ",best.trees,"
trees",sep=""),quote=FALSE)

    x.data <- eval(data[, gbm.x])                #form the temporary datasets
    y.data <- eval(data[, gbm.y])

    gbm.call <- eval(paste("gbm(y.data ~ .,n.trees = best.trees, data=x.data, verbose = F,
interaction.depth = tree.complexity,
      weights = site.weights, shrinkage = learning.rate, cv.folds = 0, distribution = as.character
(family),
      var.monotone = var.monotone, keep.data = keep.data)", sep=""))

    gbm.object <- eval(parse(text = gbm.call))

  }

#extract fitted values and summary table

  fitted.values <- predict.gbm(gbm.object,x.data,n.trees = best.trees,type="response")
  gbm.summary <- summary(gbm.object,n.trees = best.trees, plotit = FALSE)

  y_i <- y.data
  u_i <- fitted.values

  if (family == "poisson") {
    deviance.contribs <- ifelse(y_i == 0, 0, (y_i * log(y_i/u_i))) - (y_i - u_i)
    resid.deviance <- 2 * sum(deviance.contribs)
    residuals <- sqrt(abs(deviance.contribs * 2))
    residuals <- ifelse((y_i - u_i) < 0, 0 - residuals, residuals)
    u_i <- mean(y.data)
    total.deviance <- 2 * sum(ifelse(y_i == 0, 0, (y_i * log(y_i/u_i))) - (y_i - u_i))
  }

  if (family == "bernoulli") {
    deviance.contribs <- (y_i * log(u_i)) + ((1-y_i) * log(1 - u_i))
    resid.deviance <- -2 * sum(deviance.contribs)
    residuals <- sqrt(abs(deviance.contribs * 2))
    residuals <- ifelse((y_i - u_i) < 0, 0 - residuals, residuals)
```

```
    u_i <- mean(y.data)
    total.deviance <- -2 * sum((y_i * log(u_i)) + ((1-y_i) * log(1 - u_i)))
  }

  if (verbose) {
    print(paste("total deviance = ",round(total.deviance,2),sep=""),quote=F)
    print(paste("residual deviance = ",round(resid.deviance,2),sep=""),quote=F)
  }

# now assemble data to be returned

  gbm.detail <- list(dataframe = dataframe.name, gbm.x = gbm.x, predictor.names = names(x.data),
    gbm.y = gbm.y, response.name = sp.name, tree.complexity = tree.complexity, n.trees = best.trees,
    learning.rate = learning.rate, best.trees = best.trees, cv.folds = cv.folds,
    family = family, train.fraction = train.fraction, var.monotone = var.monotone )

  gbm.object$fitted <- fitted.values
  gbm.object$residuals <- residuals
  gbm.object$contributions <- gbm.summary
  gbm.object$deviances <- list(null.deviance = total.deviance, resid.deviance = resid.deviance)
  gbm.object$weights <- weights
  gbm.object$gbm.call <- gbm.detail

  rm(x.data,y.data, pos=1)           #finally, clean up the temporary dataframes

  return(gbm.object)
}

`gbm.plot` <-
function(gbm.object,                # a gbm object - could be one from gbm.step
     variable.no = 0,               # the var to plot - if zero then plots all
     smooth = FALSE,                # should we add a smoothed version of the fitted function
     rug = T,                       # plot a rug of deciles
     n.plots = length(pred.names),  # plot the first n most important preds
     common.scale = T,              # use a common scale on the y axis
     write.title = T,               # plot a title above the plot
     y.label = "fitted function",   # the default y-axis label
     x.label = var.name,            # the default x-axis label
     show.contrib = T,              # show the contribution on the x axis
     ...                            # other arguments to pass to the plotting
                                    # useful options include cex.axis, cex.lab, etc.
     )
{
# function to plot gbm response variables, with the option
# of adding a smooth representation of the response if requested
# additional options in this version allow for plotting on a common scale
# note too that fitted functions are now centered by subtracting their mean
#
# version 2.8
#
# j. leathwick/j. elith - March 2007
#

require(gbm)
require(splines)

if (length(variable.no) > 1) {stop("only one response variable can be plotted at a time")}

if (variable.no > 0) {   #we are plotting all vars in rank order of contribution
  n.plots <- 1
  }

max.vars <- length(gbm.object$contributions$var)
if (n.plots > max.vars) {
  n.plots <- max.vars
  cat("warning - reducing no of plotted predictors to maximum available (",max.vars,")\n",sep="")
  }

gbm.call <- gbm.object$gbm.call
gbm.x <- gbm.call$gbm.x
pred.names <- gbm.call$predictor.names
response.name <- gbm.call$response.name
```

```
dataframe.name <- gbm.call$dataframe
data <- eval(parse(text = dataframe.name))

predictors <- list(rep(NA,n.plots)) # matrix(0,ncol=n.plots,nrow=100)
responses <- list(rep(NA,n.plots)) # matrix(0,ncol=n.plots,nrow=100)

for (j in c(1:n.plots)) {  #cycle through the first time and get the range of the functions
  if (n.plots == 1) {
    k <- variable.no
  }
  else k <- match(gbm.object$contributions$var[j],pred.names)

  var.name <- gbm.call$predictor.names[k]
  pred.data <- data[,gbm.call$gbm.x[k]]

  response.matrix <- plot.gbm(gbm.object, k, return.grid = TRUE)

  predictors[[j]] <- response.matrix[,1]
  if (is.factor(data[,gbm.call$gbm.x[k]])) {
    predictors[[j]] <- factor(predictors[[j]],levels = levels(data[,gbm.call$gbm.x[k]]))
    }
  responses[[j]] <- response.matrix[,2] - mean(response.matrix[,2])

  if(j == 1) {
    ymin = min(responses[[j]])
    ymax = max(responses[[j]])
    }
  else {
    ymin = min(ymin,min(responses[[j]]))
    ymax = max(ymax,max(responses[[j]]))
    }
  }

  for (j in c(1:n.plots)) {
    if (n.plots == 1) {
      k <- match(pred.names[variable.no],gbm.object$contributions$var)
      if (show.contrib) {
         x.label <- paste(var.name,"  (",round(gbm.object$contributions[k,2],1),"%)",sep="")
      }
    }
    else {
      k <- match(gbm.object$contributions$var[j],pred.names)
      var.name <- gbm.call$predictor.names[k]
      if (show.contrib) {
         x.label <- paste(var.name,"  (",round(gbm.object$contributions[j,2],1),"%)",sep="")
      }
      else x.label <- var.name
    }

    if (common.scale) {
      plot(predictors[[j]],responses[[j]],ylim=c(ymin,ymax), type='l',
        xlab = x.label, ylab = y.label, ...)
    }
    else {
      plot(predictors[[j]],responses[[j]], type='l',
        xlab = x.label, ylab = y.label, ...)
    }
    if (smooth & is.vector(predictors[[j]])) {
      temp.lo <- loess(responses[[j]] ~ predictors[[j]], span = 0.3)
      lines(predictors[[j]],fitted(temp.lo), lty = 2, col = 2)
    }
    if (n.plots == 1) {
      if (write.title) {
        title(response.name)
      }
      if (rug & is.vector(data[,gbm.call$gbm.x[variable.no]])) {
        rug(quantile(data[,gbm.call$gbm.x[variable.no]], probs = seq(0, 1, 0.1), na.rm = TRUE))
      }
    }
    else {
      if (write.title & j == 1) {
        title(response.name)
```

```
      }
      if (rug & is.vector(data[,gbm.call$gbm.x[k]])) {
        rug(quantile(data[,gbm.call$gbm.x[k]], probs = seq(0, 1, 0.1), na.rm = TRUE))
      }
    }
  }
}

`gbm.perspec` <-
function(gbm.object,
     x = 1, # the first variable to be plotted
     y = 2,# the second variable to be plotted
     x.label = NULL,# allows manual specification of the x label
     y.label = NULL,# and y label
     x.range = NULL,# manual range specification for the x variable
     y.range = NULL,# and the y
     z.range = c(0,1), # allows control of the vertical axis
     pred.means = NULL,# allows specification of values for other variables
     theta = 55,# rotation
     phi=40, # and elevation
     smooth = "none", # controls smoothing of the predicted surface
     mask = FALSE,# controls masking using a sample intensity model
     perspective = TRUE,# controls whether a contour or perspective plot is drawn
     text.size=1)# allows control of text size
{
#
# gbm.perspec version 2.5 April 2006
# J Leathwick/J Elith
#
# takes a gbm boosted regression tree object produced by gbm.step and
# plots a perspective plot showing predicted values for two predictors
# as specified by number using x and y
# values for all other variables are set at their mean by default
# but values can be specified by giving a list consisting of the variable name
# and its desired value, e.g., c(name1 = 12.2, name2 = 57.6)

  require(gbm)
  require(splines)

#get the boosting model details
  gbm.call <- gbm.object$gbm.call
  gbm.x <- gbm.call$gbm.x
  n.preds <- length(gbm.x)
  gbm.y <- gbm.call$gbm.y
  pred.names <- gbm.call$predictor.names

  x.name <- gbm.call$predictor.names[x]

  if (is.null(x.label)) {
    x.label <- gbm.call$predictor.names[x]}

  y.name <- gbm.call$predictor.names[y]

  if (is.null(y.label)) {
    y.label <- gbm.call$predictor.names[y]}

  data <- eval(parse(text=gbm.call$dataframe))[,gbm.x]
  n.trees <- gbm.call$best.trees

  if (is.null(x.range)) {
    x.var <- seq(min(data[,x],na.rm=T),max(data[,x],na.rm=T),length = 50)
  }
  else {x.var <- seq(x.range[1],x.range[2],length = 50)}

  if (is.null(y.range)) {
    y.var <- seq(min(data[,y],na.rm=T),max(data[,y],na.rm=T),length = 50)
  }
  else {y.var <- seq(y.range[1],y.range[2],length = 50)}

  pred.frame <- expand.grid(list(x.var,y.var))
  names(pred.frame) <- c(x.name,y.name)
```

```
  j <- 3
  for (i in 1:n.preds) {
    if (i != x & i != y) {
      if (is.vector(data[,i])) {
        m <- match(pred.names[i],names(pred.means))
        if (is.na(m)) {
          pred.frame[,j] <- mean(data[,i],na.rm=T)
        }
        else pred.frame[,j] <- pred.means[m]
      }
      if (is.factor(data[,i])) {
        m <- match(pred.names[i],names(pred.means))
        temp.table <- table(data[,i])
        if (is.na(m)) {
          pred.frame[,j] <- rep(names(temp.table)[2],2500)
        }
        else pred.frame[,j] <- pred.means[m]
        pred.frame[,j] <- factor(pred.frame[,j],levels=names(temp.table))
      }
      names(pred.frame)[j] <- pred.names[i]
      j <- j + 1
    }
  }
#
# form the prediction
#
  prediction <- predict.gbm(gbm.object,pred.frame,n.trees = n.trees, type="response")
#
# model smooth if required
#
  if (smooth == "model") {
    pred.glm <- glm(prediction ~ ns(pred.frame[,1], df = 8) * ns(pred.frame[,2], df = 8),
data=pred.frame,family=poisson)
    prediction <- fitted(pred.glm)
  }
#
# report the maximum value
#
  max.pred <- max(prediction)
  cat("maximum value = ",round(max.pred,2),"\n")
#
# form the matrix
#
  pred.matrix <- matrix(prediction,ncol=50,nrow=50)
#
# kernel smooth if required
#
  if (smooth == "average") {  #apply a 3 x 3 smoothing average
     pred.matrix.smooth <- pred.matrix
     for (i in 2:49) {
       for (j in 2:49) {
         pred.matrix.smooth[i,j] <- mean(pred.matrix[c((i-1):(i+1)),c((j-1):(j+1))])
       }
     }
  pred.matrix <- pred.matrix.smooth
  }
#
# mask out values inside hyper-rectangle but outside of sample space
#
  if (mask) {
    mask.trees <- mask.object$gbm.call$best.trees
    point.prob <- predict.gbm(mask.object[[1]],pred.frame, n.trees = mask.trees, type="response")
    point.prob <- matrix(point.prob,ncol=50,nrow=50)
    pred.matrix[point.prob < 0.5] <- 0.0
  }
#
# and finally plot the result
#
  if (!perspective) {
     image(x = x.var, y = y.var, z = pred.matrix, zlim = c(0,50))}
  else {
  persp(x=x.var, y=y.var, z=pred.matrix, xlab = x.label, ylab = y.label, zlab = "fitted value",
```

```
    r = sqrt(10), d = 3, theta=theta,phi=phi, zlim= z.range,ticktype="detailed",
    cex=text.size, mgp = c(4,1,0))}
}

`calibration` <-
function(obs, preds, family = "binomial")
{
#
# j elith/j leathwick 17th March 2005
# calculates calibration statistics for either binomial or count data
# but the family argument must be specified for the latter
# a conditional test for the latter will catch most failures to specify
# the family
#

if (family == "bernoulli") family <- "binomial"
pred.range <- max(preds) - min(preds)
if(pred.range > 1.2 & family == "binomial") {
print(paste("range of response variable is ", round(pred.range, 2)), sep = "", quote = F)
print("check family specification", quote = F)
return()
}
if(family == "binomial") {
pred <- preds + 1e-005
pred[pred >= 1] <- 0.99999
mod <- glm(obs ~ log((pred)/(1 - (pred))), family = binomial)
lp <- log((pred)/(1 - (pred)))
a0b1 <- glm(obs ~ offset(lp) - 1, family = binomial)
miller1 <- 1 - pchisq(a0b1$deviance - mod$deviance, 2)
ab1 <- glm(obs ~ offset(lp), family = binomial)
miller2 <- 1 - pchisq(a0b1$deviance - ab1$deviance, 1)
miller3 <- 1 - pchisq(ab1$deviance - mod$deviance, 1)
}
if(family == "poisson") {
mod <- glm(obs ~ log(preds), family = poisson)
lp <- log(preds)
a0b1 <- glm(obs ~ offset(lp) - 1, family = poisson)
miller1 <- 1 - pchisq(a0b1$deviance - mod$deviance, 2)
ab1 <- glm(obs ~ offset(lp), family = poisson)
miller2 <- 1 - pchisq(a0b1$deviance - ab1$deviance, 1)
miller3 <- 1 - pchisq(ab1$deviance - mod$deviance, 1)
}
calibration.result <- c(mod$coef, miller1, miller2, miller3)
names(calibration.result) <- c("intercept", "slope", "testa0b1", "testa0|b1", "testb1|a")
return(calibration.result)
}

`roc` <-
function (obsdat, preddat)
{
# code adapted from Ferrier, Pearce and Watson's code, by J.Elith
#
# see:
# Hanley, J.A. & McNeil, B.J. (1982) The meaning and use of the area
# under a Receiver Operating Characteristic (ROC) curve.
# Radiology, 143, 29-36
#
# Pearce, J. & Ferrier, S. (2000) Evaluating the predictive performance
# of habitat models developed using logistic regression.
# Ecological Modelling, 133, 225-245.
# this is the non-parametric calculation for area under the ROC curve,
# using the fact that a MannWhitney U statistic is closely related to
# the area
#
    if (length(obsdat) != length(preddat))
        stop("obs and preds must be equal lengths")
    n.x <- length(obsdat[obsdat == 0])
    n.y <- length(obsdat[obsdat == 1])
    xy <- c(preddat[obsdat == 0], preddat[obsdat == 1])
    rnk <- rank(xy)
    wilc <- ((n.x * n.y) + ((n.x * (n.x + 1))/2) - sum(rnk[1:n.x]))/(n.x *
        n.y)
```

```
    return(round(wilc, 4))
}

`calc.deviance` <-
function(obs.values, fitted.values, weights = rep(1,length(obs.values)), family="binomial", calc.mean
= TRUE)
{
# j. leathwick/j. elith
#
# version 2.8 - 30th March 2007
#
# function to calculate deviance given two vectors of raw and fitted values
# requires a family argument which is set to binomial by default
# and weights to cater for fitting models with unequal weights
#

if (length(obs.values) != length(fitted.values))
   stop("observations and predictions must be of equal length")

y_i <- obs.values

u_i <- fitted.values

if (family == "binomial" | family == "bernoulli") {

   deviance.contribs <- (y_i * log(u_i)) + ((1-y_i) * log(1 - u_i))
   deviance <- -2 * sum(deviance.contribs * weights)

}

if (family == "poisson" | family == "Poisson") {

    deviance.contribs <- ifelse(y_i == 0, 0, (y_i * log(y_i/u_i))) - (y_i - u_i)
    deviance <- 2 * sum(deviance.contribs * weights)

}

if (family == "laplace") {
    deviance.contribs <- abs(y_i - u_i)
    deviance <- sum(deviance.contribs * weights)
    }

if (family == "gaussian") {
    deviance.contribs <- (y_i - u_i)^2
    deviance <- sum(deviance.contribs * weights)
    }

if (calc.mean) deviance <- deviance/length(obs.values)

return(deviance)

}

`gbm.interactions` <-
function(gbm.object,
   use.weights = FALSE,   # use weights for samples
   mask.object)       # a gbm object describing sample intensity
{
# functions assesses the magnitude of interaction effects in gbm models
# fitted with interaction depths greater than 1
# this is achieved by:
#   1. forming predictions on the linear scale for each predictor pair;
#   2. fitting a linear model that relates these predictions to the predictor
#        pair, with the the predictors fitted as factors;
#   3. calculating the mean value of the residuals, the magnitude of which
#        increases with the strength of any interaction effect;
#   4. results are stored in an array;
#   5. finally, the n most important interactions are identified,
#        where n is 25% of the number of interaction pairs;

  require(gbm)
```

```
  gbm.call <- gbm.object$gbm.call
  n.trees <- gbm.call$best.trees
  depth <- gbm.call$interaction.depth
  gbm.x <- gbm.call$gbm.x
  n.preds <- length(gbm.x)
  pred.names <- gbm.object$gbm.call$predictor.names
  cross.tab <- matrix(0,ncol=n.preds,nrow=n.preds)
  dimnames(cross.tab) <- list(pred.names,pred.names)

  if (use.weights) mask.trees <- mask.object$gbm.call$best.trees

  cat("Cross tabulating interactions for gbm model with ",n.preds," predictors","\n",sep="")

  data <- eval(parse(text=gbm.call$dataframe))[,gbm.x]

  for (i in 1:(n.preds - 1)) {

    if (is.vector(data[,i])) {
       x.var <- seq(min(data[,i],na.rm=T),max(data[,i],na.rm=T),length = 20)
       }
    else {
       x.var <- factor(names(table(data[,i])),levels = levels(data[,i]))
       }
    x.length <- length(x.var)

      cat(i,"\n")

      for (j in (i+1):n.preds) {

        if (is.vector(data[,j])) {
          y.var <- seq(min(data[,j],na.rm=T),max(data[,j],na.rm=T),length = 20)
         }
        else {
          y.var <- factor(names(table(data[,j])),levels = levels(data[,j]))
        }
        y.length <- length(y.var)

          pred.frame <- expand.grid(list(x.var,y.var))
          names(pred.frame) <- c(pred.names[i],pred.names[j])

          n <- 3

          for (k in 1:n.preds) {
            if (k != i & k != j) {
              if (is.vector(data[,k])) {
                pred.frame[,n] <- mean(data[,k],na.rm=T)
              }
              else {
                temp.table <- sort(table(data[,k]),decreasing = TRUE)
                pred.frame[,n] <- rep(names(temp.table)[1],x.length * y.length)
                pred.frame[,n] <- as.factor(pred.frame[,n])
              }
              names(pred.frame)[n] <- pred.names[k]
              n <- n + 1
            }
          }
#
# form the prediction
#
          prediction <- predict.gbm(gbm.object,pred.frame,n.trees = n.trees, type="link")

          if (use.weights) {
             point.prob <- predict.gbm(mask.object[[1]],pred.frame, n.trees = mask.trees,
type="response")
             interaction.test.model <- lm(prediction ~ as.factor(pred.frame[,1]) + as.factor(pred.frame
[,2]), weights = point.prob)
          }

          else {

             interaction.test.model <- lm(prediction ~ as.factor(pred.frame[,1]) + as.factor(pred.frame
[,2]))
```

```
          }

          interaction.flag <- round(mean(resid(interaction.test.model)^2) * 1000,2)

          cross.tab[i,j] <- interaction.flag

#        }   # end of x2 is vector loop
      }   # end of j loop
#    }   # end of x1 is vector loop
  }  # end of i loop

# create an index of the values in descending order

  search.index <- ((n.preds^2) + 1) - rank(cross.tab, ties.method = "first")

  n.important <- round(0.1 * ((n.preds^2)/2),0)
  var1.names <- rep(" ",n.important)
  var1.index <- rep(0,n.important)
  var2.names <- rep(" ",n.important)
  var2.index <- rep(0,n.important)
  int.size <- rep(0,n.important)

  for (i in 1:n.important) {

    index.match <- match(i,search.index)

    j <- trunc(index.match/n.preds) + 1
    var1.index[i] <- j
    var1.names[i] <- pred.names[j]

    k <- index.match%%n.preds
    if (k > 0) {   #only do this if k > 0 - otherwise we have all zeros from here on
      var2.index[i] <- k
      var2.names[i] <- pred.names[k]

      int.size[i] <- cross.tab[k,j]
    }

  }

  rank.list <- data.frame(var1.index,var1.names,var2.index,var2.names,int.size)

  return(list(rank.list = rank.list, interactions = cross.tab, gbm.call = gbm.object$gbm.call))
}

`gbm.plot.fits` <-
function(gbm.object, mask.presence = FALSE, use.factor = FALSE)
{
#
# j leathwick, j elith - 7th January 2005
#
# version 2.0 - developed in R 2.0
#
# to plot distribution of fitted values in relation to ydat from mars or other p/a models
# allows masking out of absences to enable focus on sites with high predicted values
# fitted values = those from model; raw.values = original y values
# label = text species name; ydat = predictor dataset
# mask.presence forces function to only plot fitted values for presences
# use.factor forces to use quicker printing box and whisker plot
# file.name routes to a pdf file of this name
#

    dat <- gbm.object$gbm.call$dataframe    #get the dataframe name
    dat <- as.data.frame(eval(parse(text=dat)))   #and now the data

    n.cases <- nrow(dat)

    gbm.call <- gbm.object$gbm.call#and the mars call details
    gbm.x <- gbm.call$gbm.x
    gbm.y <- gbm.call$gbm.y
    family <- gbm.call$family
```

```
    xdat <- as.data.frame(dat[,gbm.x])
    ydat <- as.data.frame(dat[,gbm.y])

    n.preds <- ncol(xdat)

    fitted.values <- gbm.object$fitted

    pred.names <- names(dat)[gbm.x]
    sp.name <- names(dat)[gbm.y]

    if (mask.presence) {
  mask <- ydat == 1 }
    else {
  mask <- rep(TRUE, length = n.cases) }

    robust.max.fit <- approx(ppoints(fitted.values[mask]), sort(fitted.values[mask]), 0.99) #find 99%
ile value

    for (j in 1:n.preds) {

  if (is.numeric(xdat[mask,j])) {
            wt.mean <- zapsmall(mean((xdat[mask, j] * fitted.values[mask]^5)/mean(fitted.values[mask]
^5),na.rm=TRUE),2)
            }
        else {wt.mean <- "na"}
   if (use.factor) {
      temp <- factor(cut(xdat[mask, j], breaks = 12))
if (family == "binomial") {
    plot(temp, fitted.values[mask], xlab = pred.names[j], ylab = "fitted values", ylim = c(0, 1))}
else {
    plot(temp, fitted.values[mask], xlab = pred.names[j], ylab = "fitted values")}
}
   else {
if (family == "binomial") {
    plot(xdat[mask, j], fitted.values[mask], xlab = pred.names[j], ylab = "fitted values",
ylim = c(0, 1))}
    else {
plot(xdat[mask, j], fitted.values[mask], xlab = pred.names[j], ylab = "fitted values")}
    }
abline(h = (0.333 * robust.max.fit$y), lty = 2.)
if (j == 1) {
title(paste(sp.name, ", wtm = ", wt.mean))}
else {
title(paste("wtm = ", wt.mean))}
    }
}

`gbm.simplify` <-
function(gbm.object,          # a gbm object describing sample intensity
  n.folds = 10,               # number of times to repeat the analysis
  n.drops = "auto",           # can be automatic or an integer specifying the number of drops to check
  alpha = 1,                  # controls stopping when n.drops = "auto"
  prev.stratify = TRUE,       # use prevalence stratification in selecting evaluation data
  eval.data = NULL,           # an independent evaluation data set - leave here for now
  plot = TRUE)                # plot results
{
# function to simplify a brt model fitted using gbm.step
#
# version 2.7 - J. Leathwick/J. Elith - June 2006
#
# starts with an inital cross-validated model as produced by gbm.step
# and then assesses the potential to remove predictors using k-fold cv
# does this for each fold, removing the lowest contributing predictor,
# and repeating this process for a set number of steps
# after the removal of each predictor, the change in predictive deviance
# is computed relative to that obtained when using all predictors
# it returns a list containing the mean change in deviance and its se
# as a function of the number of variables removed
# a table contains a summary of the order in which variables are removed
# and a list consisting of the variables retained at each step in the
# variable removal process - the latter can be used as an argument to gbm.step
# e.g., gbm.step(data = data, gbm.x = simplify.object$pred.list[[4]]...
```

```
# would implement a new analysis with the original predictor set, minus its
# four lowest contributing predictors
#

require(gbm)

# first get the original analysis details..

  gbm.call <- gbm.object$gbm.call
  data <- eval(parse(text=gbm.call$dataframe))
  n.cases <- nrow(data)
  gbm.x <- gbm.call$gbm.x
  gbm.y <- gbm.call$gbm.y
  family <- gbm.call$family
  lr <- gbm.call$learning.rate
  tc <- gbm.call$tree.complexity
  start.preds <- length(gbm.x)
  max.drops <- start.preds - 2
  response.name <- gbm.call$response.name
  predictor.names <- gbm.call$predictor.names
  n.trees <- gbm.call$best.trees
  pred.list <- list(initial = gbm.x)

  if (n.drops == "auto") auto.stop <- TRUE

  else auto.stop <- FALSE

# take a copy of the original data and starting predictors

  orig.data <- data
  orig.gbm.x <- gbm.x

#  if (!is.null(eval.data)) independent.test <- TRUE
#    else independent.test <- FALSE

# extract original performance statistics...

  original.deviance <- round(gbm.object$cv.statistics$deviance.mean,4)
  original.deviance.se <- round(gbm.object$cv.statistics$deviance.se,4)

  cat("gbm.simplify - version 2.6","\n\n")
  cat("simplifying gbm.step model for ",response.name," with ",start.preds," predictors",sep="")
  cat(" and ",n.cases," observations \n",sep="")
  cat("original deviance = ",original.deviance,"(",original.deviance.se,")\n\n",sep="")

# check that n.drops is less than n.preds - 2 and update if required

  if (auto.stop) {
    cat("variable removal will proceed until average change exceeds the original se\n\n")
    n.drops <- 1 }
  else{
    if (n.drops > start.preds - 2) {
      cat("value of n.drops (",n.drops,") is greater than permitted","\n",
        "resetting value to ",start.preds - 2,"\n\n",sep="")
      n.drops <- start.preds - 2
    }
    else {
      cat("a fixed number of",n.drops,"drops will be tested\n\n")
    }
  }

# set up storage for results

  dev.results <- matrix(0, nrow = n.drops, ncol = n.folds)
  dimnames(dev.results) <- list(paste("drop.",1:n.drops,sep=""),
   paste("rep.",1:n.folds,sep=""))

  drop.count <- matrix(NA, nrow = start.preds, ncol = n.folds)
  dimnames(drop.count) <- list(predictor.names,paste("rep.",1:n.folds,sep=""))

  original.deviances <- rep(0,n.folds)
```

```
  model.list <- list(paste("model",c(1:n.folds),sep=""))     # dummy list for the tree models

# create gbm.fixed function call

  gbm.call.string <- paste("try(gbm.fixed(data=train.data,gbm.x=gbm.new.x,gbm.y=gbm.y,",sep="")
  gbm.call.string <- paste(gbm.call.string,"family=family,learning.rate=lr,tree.complexity=tc,",sep="")
  gbm.call.string <- paste(gbm.call.string,"n.trees = ",n.trees,",verbose=FALSE))",sep="")

# now set up the fold structure

  if (prev.stratify & family == "bernoulli") {
    presence.mask <- data[,gbm.y] == 1
    absence.mask <- data[,gbm.y] == 0
    n.pres <- sum(presence.mask)
    n.abs <- sum(absence.mask)

# create a vector of randomised numbers and feed into presences
    selector <- rep(0,n.cases)
    temp <- rep(seq(1, n.folds, by = 1), length = n.pres)
    temp <- temp[order(runif(n.pres, 1, 100))]
    selector[presence.mask] <- temp

# and then do the same for absences
    temp <- rep(seq(1, n.folds, by = 1), length = n.abs)
    temp <- temp[order(runif(n.abs, 1, 100))]
    selector[absence.mask] <- temp
    }

  else {  #otherwise make them random with respect to presence/absence
    selector <- rep(seq(1, n.folds, by = 1), length = n.cases)
    selector <- selector[order(runif(n.cases, 1, 100))]
    }

# now start by creating the intial models for each fold

  cat("creating initial models...\n\n")

  gbm.new.x <- orig.gbm.x

  for (i in 1:n.folds) {

# create the training and prediction folds

    train.data <- orig.data[selector!=i,]
    eval.data <- orig.data[selector==i,]

    model.list[[i]] <- eval(parse(text=gbm.call.string))  # create a fixed size object

# now make predictions to the withheld fold

    u_i <- eval.data[,gbm.y]
    y_i <- predict.gbm(model.list[[i]], eval.data, n.trees, "response")

    original.deviances[i] <- round(calc.deviance(u_i,y_i, family = family, calc.mean = TRUE),4)

  } # end of creating initial models

  n.steps <- 1

  while (n.steps <= n.drops & n.steps <= max.drops) {

    cat("dropping predictor",n.steps,"\n")

    for (i in 1:n.folds) {

# get the right data

    train.data <- orig.data[selector!=i,]
    eval.data <- orig.data[selector==i,]

# get the current model details
```

```
    gbm.call <- model.list[[i]]$gbm.call
    gbm.x <- gbm.call$gbm.x
    n.preds <- length(gbm.x)
    these.pred.names <- model.list[[i]]$gbm.call$predictor.names
    contributions <- model.list[[i]]$contributions

# get the index number in pred.names of the last variable in the contribution table

    last.variable <- match(as.character(contributions[n.preds,1]),these.pred.names)
    gbm.new.x <- gbm.x[-last.variable]

# and keep a record of what has been dropped

    last.variable <- match(as.character(contributions[n.preds,1]),predictor.names)
    drop.count[last.variable,i] <- n.steps

    model.list[[i]] <- eval(parse(text=gbm.call.string))  # create a fixed size object

    u_i <- eval.data[,gbm.y]
    y_i <- predict.gbm(model.list[[i]],eval.data,n.trees,"response")

    deviance <- round(calc.deviance(u_i,y_i, family = family, calc.mean = TRUE),4)

# calculate difference between intial and new model by subtracting new from old because we want to
minimise deviance

    dev.results[n.steps,i] <- round(deviance - original.deviances[i] ,4)

    }

  if (auto.stop){ # check to see if delta mean is less than original deviance error estimate

    delta.mean <- mean(dev.results[n.steps,])

    if (delta.mean < (alpha * original.deviance.se)) {
      n.drops <- n.drops + 1
      dev.results <- rbind(dev.results, rep(0,n.folds))
      }
    }
  n.steps <- n.steps + 1
  }

# now label the deviance matrix

  dimnames(dev.results) <- list(paste("drop.",1:n.drops,sep=""),
   paste("rep.",1:n.folds,sep=""))

# calculate mean changes in deviance and their se

  mean.delta <- apply(dev.results,1,mean)
  se.delta <- sqrt(apply(dev.results,1,var))/sqrt(n.folds)

# make a list of variables to remove

  pred.mean.score <- apply(drop.count,1,mean,na.rm=T)
  removal.list <- names(sort(pred.mean.score))

#and then the corresponding numbers

  n.remove <- length(removal.list)
  removal.numbers <- rep(0,n.remove)

# construct predictor lists to faciliate final model fitting

  for (i in 1:n.drops) {
    removal.numbers[i] <- match(removal.list[i],predictor.names)
    pred.list[[i]] <- orig.gbm.x[0-removal.numbers[1:i]]
    names(pred.list)[i] <- paste("preds.",i,sep="")
  }

  removal.summary <- data.frame(pred.no = removal.numbers, removal.score = sort(pred.mean.score))
```

```
  if (plot) {
    y.max <- 1.5 * max(mean.delta + se.delta)
    y.min <- 1.5 * min(mean.delta - se.delta)
    plot(seq(1,n.drops),mean.delta,xlab="variables removed",
      ylab = "predictive deviance",type='l',ylim=c(y.min,y.max))
    lines(seq(1,n.drops),mean.delta + se.delta,lty = 2)
    lines(seq(1,n.drops),mean.delta - se.delta,lty = 2)
    abline(h = 0 , lty = 2, col = 3)
    min.y <- min(mean.delta)
    min.pos <- match(min.y,mean.delta)
    abline(v = min.pos, lty = 3, col = 2)
    title(paste("RFE deviance - ",response.name," - folds = ",n.folds,sep=""))
  }

  return(list(delta.mean = mean.delta, delta.se = se.delta,
    dev.matrix = dev.results, drop.count = drop.count, pred.list = pred.list,
    removal.summary = removal.summary,
    gbm.call = gbm.call))

}

"gbm.predict.grids" <-
function(model, new.dat, want.grids = F, preds2R = T, sp.name = "preds",pred.vec = NULL, filepath =
NULL,

num.col = NULL, num.row = NULL, xll = NULL, yll = NULL, cell.size = NULL, no.data = NULL, plot=F,
full.grid=T, part.number=NULL, part.row = NULL, header = T)
{
# J.Elith / J.Leathwick, March 07
# to make predictions to sites or grids. If to sites, the
# predictions are written to the R workspace. If to grid,
# the grids are written to a nominated directory and optionally also
# plotted in R and written to the workspace
#
# new data (new.dat) must be a data frame with column names identical
# to names for all variables in the model used for prediction
#
# pred.vec is a vector of -9999's, the length of the scanned full grid
# (i.e. without nodata values excluded)
#
# filepath must specify the whole path as a character vector,but without the final file
# name - eg "c:/gbm/"

temp <- predict.gbm(model, new.dat, n.trees=model$gbm.call$best.trees, type="response")

if(want.grids)
{
newname <- paste(filepath, sp.name,".asc", sep="")
full.pred <- pred.vec
full.pred[as.numeric(row.names(new.dat))] <- temp
if(header){
write(paste("ncols          ",num.col,sep=""),newname)
write(paste("nrows          ",num.row,sep=""),newname,append=T)
write(paste("xllcorner      ",xll,sep=""),newname,append=T)
write(paste("yllcorner      ",yll,sep=""),newname,append=T)
write(paste("cellsize       ",cell.size,sep=""),newname,append=T)
write(paste("NODATA_value ",no.data,sep=""),newname,append=T)
}

        if(full.grid){
         full.pred.mat <- matrix(full.pred, nrow=num.row, ncol=num.col, byrow=T)
        if (plot)
        {
        image(z = t(full.pred.mat)[, nrow(full.pred.mat):1], zlim =  c(0,1), col = rev(topo.colors
(12)))
        }


        write.table(full.pred.mat, newname, sep=" ", append=T, row.names=F, col.names=F)

        #also write to R directory, if required:
```

```
        if(preds2R){assign(sp.name,temp, pos=1)}
        }

        else{
         full.pred.mat <- matrix(full.pred, nrow=part.row, ncol=num.col, byrow=T)
        write.table(full.pred.mat, newname, sep=" ", append=T, row.names=F, col.names=F)
        if(preds2R){assign(paste(sp.name, part.number, sep=""),temp, pos=1)}
        }

}

else{
assign(sp.name,temp, pos=1)
}

}
```