1. Use the below given data set

DataSet

2. Perform the below given activities:

a. Create classification model using different classifiers

b. Verify model goodness of fit

c. Apply all the model validation techniques.

|  |
| --- |
| options(digits=3) |
|  |  |
|  | #' Load the data set |
|  | data(Zoo, package="mlbench") |
|  | head(Zoo) |
|  |  |
|  |  |
|  | #' Use multi-core support for cross-validation. |
|  | #' \_\_Note:\_\_ Does not work with rJava used in RWeka below. |
|  | #library(doParallel) |
|  | #registerDoParallel() |
|  | #getDoParWorkers() |
|  |  |
|  | #' # Fitting Different Classification Models |
|  | #' |
|  | #' Load the caret data mining package |
|  | library(caret) |
|  |  |
|  | #' Create fixed sampling scheme (10-folds) so we can compare the models |
|  | #' later on. |
|  | train <- createFolds(Zoo$type, k=10) |
|  |  |
|  |  |
|  | #' For help with building models in caret see: ? train |
|  | #' |
|  | #' \_\_Note:\_\_ Be careful if you have many `NA` values in your data. `train()` |
|  | #' and cross-validation many fail in some cases. If that is the case then you |
|  | #' can remove features (columns) which have many `NA`s, omit `NA`s using |
|  | #' `na.omit()` or use imputation to replace them with reasonable |
|  | #' values (e.g., by the feature mean or via kNN). |
|  | #' |
|  |  |
|  | #' ## Conditional Inference Tree (Decision Tree) |
|  | ctreeFit <- train(type ~ ., method = "ctree", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train)) |
|  | ctreeFit |
|  | plot(ctreeFit$finalModel) |
|  |  |
|  | #' The final model can be directly used for predict() |
|  | predict(ctreeFit, Zoo[1:2,]) |
|  |  |
|  | #' ## C 4.5 Decision Tree |
|  | library(RWeka) |
|  | C45Fit <- train(type ~ ., method = "J48", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train)) |
|  | C45Fit |
|  | C45Fit$finalModel |
|  |  |
|  | #' ## K-Nearest Neighbors |
|  | #' |
|  | #' \_\_Note:\_\_ kNN uses Euclidean distance, so data should be standardized (scaled) first. |
|  | #' Here legs are measured between 0 and 6 while all other variables are between |
|  | #' 0 and 1. |
|  | Zoo\_scaled <- cbind(as.data.frame(scale(Zoo[,-17])), type = Zoo[,17]) |
|  | knnFit <- train(type ~ ., method = "knn", data = Zoo\_scaled, |
|  | tuneLength = 5, tuneGrid=data.frame(k=1:10), |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train)) |
|  | knnFit |
|  | knnFit$finalModel |
|  |  |
|  | #' ## PART (Rule-based classifier) |
|  | rulesFit <- train(type ~ ., method = "PART", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train)) |
|  | rulesFit |
|  | rulesFit$finalModel |
|  |  |
|  |  |
|  | #' ## Linear Support Vector Machines |
|  | svmFit <- train(type ~., method = "svmLinear", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train)) |
|  | svmFit |
|  | svmFit$finalModel |
|  |  |
|  | #' ## Random Forest |
|  | randomForestFit <- train(type ~ ., method = "rf", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train)) |
|  | randomForestFit |
|  | randomForestFit$finalModel |
|  |  |
|  |  |
|  | #' ## Gradient Boosted Decision Trees (xgboost) |
|  | xgboostFit <- train(type ~ ., method = "xgbTree", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train), |
|  | tuneGrid = expand.grid( |
|  | nrounds = 20, |
|  | max\_depth = 3, |
|  | colsample\_bytree = .6, |
|  | eta = 0.1, |
|  | gamma=0, |
|  | min\_child\_weight = 1, |
|  | subsample = .5 |
|  | )) |
|  | xgboostFit |
|  | xgboostFit$finalModel |
|  |  |
|  |  |
|  | #' ## Artificial Neural Network |
|  | nnetFit <- train(type ~ ., method = "nnet", data = Zoo, |
|  | tuneLength = 5, |
|  | trControl = trainControl( |
|  | method = "cv", indexOut = train), |
|  | trace = FALSE) |
|  | nnetFit |
|  | nnetFit$finalModel |
|  |  |
|  | #' # Compare Models |
|  | resamps <- resamples(list( |
|  | ctree=ctreeFit, |
|  | C45=C45Fit, |
|  | SVM=svmFit, |
|  | KNN=knnFit, |
|  | rules=rulesFit, |
|  | randomForest=randomForestFit, |
|  | xgboost=xgboostFit, |
|  | NeuralNet=nnetFit |
|  | )) |
|  | resamps |
|  | summary(resamps) |
|  |  |
|  | difs <- diff(resamps) |
|  | difs |
|  | summary(difs) |

|  |
| --- |
|  |
| setMethod("fit.gstatModel", signature(observations = "SpatialPointsDataFrame", formulaString = "formula", covariates = "SpatialPixelsDataFrame"), function(observations, formulaString, covariates, method = list("GLM", "rpart", "randomForest", "quantregForest", "xgboost", "ranger"), dimensions = list("2D", "3D", "2D+T", "3D+T"), fit.family = gaussian(), stepwise = TRUE, vgmFun = "Exp", subsample = 5000, subsample.reg = 10000, ...){ |
|  |  |
|  | method <- method[[1]] |
|  | dimensions <- dimensions[[1]] |
|  | ## TH: the function only works with 2D maps at the moment: |
|  | if(length(attr(coordinates(observations), "dimnames")[[2]])>2){ |
|  | warning("This method uses only 2D coordinates of the points. For 3D data consider using the 'geosamples-class'.") |
|  | } |
|  | ## TH: the model has to include at least one covariate: |
|  | if(length(all.vars(formulaString)[-1])==0){ |
|  | stop("No covariates have been specified in the 'formulaString'") |
|  | } |
|  | ## check argument 'fit.family' |
|  | if(!missing(fit.family) & !method=="GLM"){ warning("'fit.family' argument will be ignored. Using 'method=\"GLM\"' instead.") } |
|  |  |
|  | ## overlay: |
|  | ov <- over(observations, covariates) |
|  | ## all variables of interest: |
|  | tv <- all.vars(formulaString)[1] |
|  | seln <- names(covariates) %in% all.vars(formulaString)[-1] |
|  | ## check if all covariates are available: |
|  | if(sum(!is.na(seln))==0){ |
|  | stop("None of the covariates in the 'formulaString' matches the names in the 'covariates' object") |
|  | } |
|  | ov <- cbind(data.frame(observations[,tv]), ov) |
|  |  |
|  | ## copy coordinate column names for consistency: |
|  | xyn <- which(names(ov) %in% attr(observations@bbox, "dimnames")[[1]]) |
|  | if(is.null(attr(covariates@bbox, "dimnames")[[1]])) { |
|  | dim.names = attr(observations@bbox, "dimnames")[[1]] |
|  | } else { |
|  | dim.names = attr(covariates@bbox, "dimnames")[[1]] |
|  | } |
|  | if(length(xyn)==2) { |
|  | names(ov)[xyn] <- dim.names[1:2] |
|  | } else { |
|  | names(ov)[xyn] <- dim.names |
|  | } |
|  |  |
|  | ## check the size of output: |
|  | if(nrow(ov)==0|is.null(ov[,tv])) { |
|  | stop("The 'over' operations resulted in an empty set.") |
|  | } |
|  |  |
|  | ## fit/filter the regression model: |
|  | m <- fit.regModel(formulaString = formulaString, rmatrix = ov, predictionDomain = covariates[seln], method = method, dimensions = dimensions, fit.family = fit.family, stepwise = stepwise, vgmFun = vgmFun, subsample = subsample, subsample.reg = subsample.reg, ...) |
|  | return(m) |
|  |  |
|  | }) |
|  |  |
|  |  |
|  | ## Fit a RK model using geosamples class: |
|  | setMethod("fit.gstatModel", signature(observations = "geosamples", formulaString = "formula", covariates = "SpatialPixelsDataFrame"), function(observations, formulaString, covariates, method = list("GLM", "rpart", "randomForest", "quantregForest", "xgboost", "ranger"), dimensions = list("2D", "3D", "2D+T", "3D+T"), fit.family = gaussian(), stepwise = TRUE, vgmFun = "Exp", subsample = 5000, subsample.reg = 10000, ...){ |
|  |  |
|  | method <- method[[1]] |
|  | dimensions <- dimensions[[2]] |
|  | ## TH: the model has to include at least one covariate: |
|  | if(length(all.vars(formulaString)[-1])==0){ |
|  | stop("No covariates have been specified in the 'formulaString'") |
|  | } |
|  |  |
|  | ## all columns of interest: |
|  | methodid <- all.vars(formulaString)[1] |
|  | seln <- names(covariates) %in% all.vars(formulaString)[-1] |
|  | xyn <- attr(covariates@bbox, "dimnames")[[1]] |
|  | ## check if all covariates are available: |
|  | if(sum(!is.na(seln))==0){ |
|  | stop("None of the covariates in the 'formulaString' matches the names in the 'covariates' object") |
|  | } |
|  |  |
|  | ## prepare regression matrix: |
|  | ov <- over(x=covariates, y=observations, method=methodid, var.type = "numeric") |
|  | if(nrow(ov)==0|is.null(ov$observedValue)) { |
|  | warning("The 'over' operations resulted in an empty set. Check 'methodid' column.") |
|  | } |
|  | ## geostats only possible with numeric variables: |
|  | ov[,methodid] = as.numeric(ov$observedValue) |
|  | ov$observedValue = NULL |
|  |  |
|  | ## subset to columns of interest: |
|  | if(dimensions == "3D"){ ov <- ov[, c(methodid, names(covariates)[seln], xyn, "altitude")] } |
|  | if(dimensions == "2D"){ ov <- ov[, c(methodid, names(covariates)[seln], xyn)] } |
|  |  |
|  | ## fit/filter the regression model: |
|  | m <- fit.regModel(formulaString = formulaString, rmatrix = ov, predictionDomain = covariates[seln], method = method, dimensions = dimensions, fit.family = fit.family, stepwise = stepwise, vgmFun = vgmFun, subsample = subsample, subsample.reg = subsample.reg, ...) |
|  | return(m) |
|  |  |
|  | }) |
|  |  |
|  |  |
|  | ## Fit a RK model to a list of covariates / formulas: |
|  | setMethod("fit.gstatModel", signature(observations = "geosamples", formulaString = "list", covariates = "list"), function(observations, formulaString, covariates, method = list("GLM", "rpart", "randomForest", "quantregForest", "xgboost", "ranger"), dimensions = list("2D", "3D", "2D+T", "3D+T"), fit.family = gaussian(), stepwise = TRUE, vgmFun = "Exp", subsample = 5000, subsample.reg = 10000, ...){ |
|  |  |
|  | method <- method[[1]] |
|  | dimensions <- dimensions[[2]] |
|  | if(!length(formulaString)==length(covariates)){ |
|  | stop("'formulaString' and 'covariates' lists of same size expected") |
|  | } |
|  |  |
|  | rkm.l <- list(NULL) |
|  | for(l in 1:length(covariates)){ |
|  | rkm.l[[l]] <- fit.gstatModel(observations, formulaString[[l]], covariates[[l]], method = method, fit.family = fit.family, stepwise = stepwise, subsample = subsample, subsample.reg = subsample.reg, dimensions = dimensions, ...) |
|  | } |
|  | return(rkm.l) |
|  |  |
|  | }) |
|  |  |
|  |  |
|  | ## Fit a RK model and return an object of class "gstatModel" for a list of multiscale grids: |
|  | setMethod("fit.gstatModel", signature(observations = "geosamples", formulaString = "formula", covariates = "list"), function(observations, formulaString, covariates, method = list("GLM", "rpart", "randomForest", "quantregForest", "xgboost", "ranger"), dimensions = list("2D", "3D", "2D+T", "3D+T"), fit.family = gaussian(), stepwise = TRUE, vgmFun = "Exp", subsample = 5000, subsample.reg = 10000, ...){ |
|  |  |
|  | method <- method[[1]] |
|  | dimensions <- dimensions[[2]] |
|  | if(!any(sapply(covariates, class)=="SpatialPixelsDataFrame")){ |
|  | stop("List of covariates of class 'SpatialPixelsDataFrame' expected") |
|  | } |
|  |  |
|  | if(!missing(fit.family) & !method=="GLM"){ |
|  | warning("'fit.family' argument will be ignored. Use 'method=\"GLM\"' instead.") |
|  | } |
|  |  |
|  | ## covariate names: |
|  | covs <- unlist(sapply(covariates, FUN=function(x){names(x)})) |
|  | if(!length(unique(covs))==length(covs)){ stop("'Covariates' column names must be unique") } |
|  |  |
|  | ## target variable: |
|  | methodid <- all.vars(formulaString)[1] |
|  |  |
|  | ## prepare regression matrix: |
|  | ov <- list(NULL) |
|  | for(j in 1:length(covariates)){ |
|  | ov[[j]] <- over(x=covariates[[j]], y=observations, method=methodid, var.type="numeric") |
|  | if(nrow(ov[[j]])==0|is.null(ov[[j]]$observedValue)) { |
|  | warning("The 'over' operations resulted in an empty set. Check 'methodid' column.") |
|  | } |
|  | } |
|  | xyn = attr(covariates[[1]]@bbox, "dimnames")[[1]] |
|  | ## coordinates in the local space: |
|  | xy <- ov[[1]][,xyn] |
|  |  |
|  | gn <- names(observations@data)[!(names(observations@data) %in% c("observationid", xyn))] |
|  | ## merge all regression matrices: |
|  | ov <- Reduce(function(x,y) {merge(x[!(names(x) %in% gn)],y[!(names(y) %in% gn)], by="observationid")}, ov) |
|  | ov <- merge(observations@data, ov[,c("observationid", covs)], by="observationid") |
|  | ov <- cbind(ov, xy) |
|  |  |
|  | ## geostats only possible with numeric variables: |
|  | ov[,methodid] <- as.numeric(ov$observedValue) |
|  | ov$observedValue <- NULL |
|  |  |
|  | ## check the size of the output object: |
|  | if(nrow(ov)==0|is.null(ov[,methodid])) { |
|  | warning("The 'over' operations resulted in an empty set. Check 'methodid' column.") |
|  | } |
|  |  |
|  | ## fit/filter the regression model: |
|  | m <- fit.regModel(formulaString = formulaString, rmatrix = ov, predictionDomain = covariates[[1]], method = method, dimensions = dimensions, fit.family = fit.family, stepwise = stepwise, vgmFun = vgmFun, subsample = subsample, subsample.reg = subsample.reg, ...) |
|  | return(m) |
|  |  |
|  | }) |
|  |  |
|  |  |
|  | "print.gstatModel" <- function(x, ...){ |
|  | print(x@regModel) |
|  | print(x@vgmModel) |
|  | summary(x@sp) |
|  | } |
|  |  |
|  | ## BK: plot functionality; see also ?plotKML::plot.SpatialPredictions |
|  | "plot.gstatModel" <- function(x, ...){ |
|  | dev.new(width=9, height=5) |
|  | par(mfrow=c(1,2)) |
|  | if(any(class(x@regModel) == "lm")|any(class(x@regModel)=="quantregForest")|any(class(x@regModel)=="randomForest")|any(class(x@regModel)=="rpart")|any(class(x@regModel)=="ranger")|any(class(x@regModel)=="xgboost")){ |
|  | if(any(class(x@regModel) == "lm")){ |
|  | plot(y=fitted(x@regModel), x=x@regModel$y, col = alpha("grey18", 0.6), pch=16, xlab='observed', ylab='predicted', main='Goodness of fit', asp=1, xlim=range(x@regModel$y), ylim=range(x@regModel$y), ...) |
|  | } |
|  | if(class(x@regModel)[1]=="quantregForest"){ |
|  | plot(y=x@regModel$predicted, x=x@regModel$y, col = alpha("grey18", 0.6), pch=16, xlab='observed', ylab='predicted', main='Goodness of fit', asp=1, xlim=range(x@regModel$y), ylim=range(x@regModel$y), ...) |
|  | } else { |
|  | if(any(class(x@regModel)=="randomForest")|any(class(x@regModel)=="rpart")|any(class(x@regModel)=="xgboost")){ |
|  | plot(y=predict(x@regModel), x=x@regModel$y, col = alpha("grey18", 0.6), pch=16, xlab='observed', ylab='predicted', main='Goodness of fit', asp=1, xlim=range(x@regModel$y), ylim=range(x@regModel$y), ...) |
|  | } |
|  | if(any(class(x@regModel)=="ranger")){ |
|  | plot(y=predict(x@regModel, x@sp@data)$predictions, x=x@sp@data[,1], col = alpha("grey18", 0.6), pch=16, xlab='observed', ylab='predicted', main='Goodness of fit', asp=1, xlim=range(x@sp@data[,1]), ylim=range(x@sp@data[,1]), ...) |
|  | } |
|  | } |
|  | } else { |
|  | pred <- fitted(x@regModel) |
|  | obs <- pred+resid(x@regModel) |
|  | plot(y=pred, x=obs, col = alpha("grey18", 0.6), pch=16, xlab='observed', ylab='predicted', main='Goodness of fit', asp=1, xlim=range(obs), ylim=range(obs), ...) |
|  | } |
|  | abline(a=0, b=1, lwd=2, lty=2, col="green") |
|  | vgmmodel <- x@vgmModel |
|  | class(vgmmodel) <- c("variogramModel","data.frame") |
|  | plot(x=x@svgmModel$dist, y=x@svgmModel$gamma, pch="+", col = "grey18", xlab='distance', cex=1.1, ylab='gamma', ylim = c(0, max(x@svgmModel$gamma)), main='Residual variogram') |
|  | vline <- variogramLine(vgmmodel, maxdist=max(x@svgmModel$dist), n=length(x@svgmModel$dist)) |
|  | lines(x=vline$dist, y=vline$gamma, col="green", lwd=2) |
|  | } |
|  |  |
|  | setMethod("plot", signature("gstatModel"), plot.gstatModel) |
|  |  |

|  |
| --- |
| #' Control parameters for train |
|  | #' |
|  | #' Control the computational nuances of the \code{\link{train}} function |
|  | #' |
|  | #' When setting the seeds manually, the number of models being evaluated is |
|  | #' required. This may not be obvious as \code{train} does some optimizations |
|  | #' for certain models. For example, when tuning over PLS model, the only model |
|  | #' that is fit is the one with the largest number of components. So if the |
|  | #' model is being tuned over \code{comp in 1:10}, the only model fit is |
|  | #' \code{ncomp = 10}. However, if the vector of integers used in the |
|  | #' \code{seeds} arguments is longer than actually needed, no error is thrown. |
|  | #' |
|  | #' Using \code{method = "none"} and specifying more than one model in |
|  | #' \code{\link{train}}'s \code{tuneGrid} or \code{tuneLength} arguments will |
|  | #' result in an error. |
|  | #' |
|  | #' Using adaptive resampling when \code{method} is either \code{"adaptive\_cv"}, |
|  | #' \code{"adaptive\_boot"} or \code{"adaptive\_LGOCV"}, the full set of resamples |
|  | #' is not run for each model. As resampling continues, a futility analysis is |
|  | #' conducted and models with a low probability of being optimal are removed. |
|  | #' These features are experimental. See Kuhn (2014) for more details. The |
|  | #' options for this procedure are: |
|  | #' |
|  | #' \itemize{ \item \code{min}: the minimum number of resamples used before |
|  | #' models are removed \item \code{alpha}: the confidence level of the one-sided |
|  | #' intervals used to measure futility \item \code{method}: either generalized |
|  | #' least squares (\code{method = "gls"}) or a Bradley-Terry model (\code{method |
|  | #' = "BT"}) \item \code{complete}: if a single parameter value is found before |
|  | #' the end of resampling, should the full set of resamples be computed for that |
|  | #' parameter. ) } |
|  | #' |
|  | #' The option \code{search = "grid"} uses the default grid search routine. When |
|  | #' \code{search = "random"}, a random search procedure is used (Bergstra and |
|  | #' Bengio, 2012). See \url{http://topepo.github.io/caret/random-hyperparameter-search.html} for |
|  | #' details and an example. |
|  | #' |
|  | #' The supported bootstrap methods are: |
|  | #' |
|  | #' \itemize{ |
|  | #' \item \code{"boot"}: the usual bootstrap. |
|  | #' \item \code{"boot632"}: the 0.632 bootstrap estimator (Efron, 1983). |
|  | #' \item \code{"optimism\_boot"}: the optimism bootstrap estimator. |
|  | #' (Efron and Tibshirani, 1994). |
|  | #' \item \code{"boot\_all"}: all of the above (for efficiency, |
|  | #' but "boot" will be used for calculations). |
|  | #' } |
|  | #' |
|  | #' The \code{"boot632"} method should not to be confused with the 0.632+ |
|  | #' estimator proposed later by the same author. |
|  | #' |
|  | #' Note that if \code{index} or \code{indexOut} are specified, the label shown by \code{train} may not be accurate since these arguments supersede the \code{method} argument. |
|  | #' |
|  | #' @param method The resampling method: \code{"boot"}, \code{"boot632"}, |
|  | #' \code{"optimism\_boot"}, \code{"boot\_all"}, |
|  | #' \code{"cv"}, \code{"repeatedcv"}, \code{"LOOCV"}, \code{"LGOCV"} (for |
|  | #' repeated training/test splits), \code{"none"} (only fits one model to the |
|  | #' entire training set), \code{"oob"} (only for random forest, bagged trees, |
|  | #' bagged earth, bagged flexible discriminant analysis, or conditional tree |
|  | #' forest models), \code{timeslice}, \code{"adaptive\_cv"}, \code{"adaptive\_boot"} or |
|  | #' \code{"adaptive\_LGOCV"} |
|  | #' @param number Either the number of folds or number of resampling iterations |
|  | #' @param repeats For repeated k-fold cross-validation only: the number of |
|  | #' complete sets of folds to compute |
|  | #' @param verboseIter A logical for printing a training log. |
|  | #' @param returnData A logical for saving the data |
|  | #' @param returnResamp A character string indicating how much of the resampled |
|  | #' summary metrics should be saved. Values can be \code{"final"}, \code{"all"} |
|  | #' or \code{"none"} |
|  | #' @param savePredictions an indicator of how much of the hold-out predictions |
|  | #' for each resample should be saved. Values can be either \code{"all"}, |
|  | #' \code{"final"}, or \code{"none"}. A logical value can also be used that |
|  | #' convert to \code{"all"} (for true) or \code{"none"} (for false). |
|  | #' \code{"final"} saves the predictions for the optimal tuning parameters. |
|  | #' @param p For leave-group out cross-validation: the training percentage |
|  | #' @param search Either \code{"grid"} or \code{"random"}, describing how the |
|  | #' tuning parameter grid is determined. See details below. |
|  | #' @param initialWindow,horizon,fixedWindow,skip possible arguments to |
|  | #' \code{\link{createTimeSlices}} when method is \code{timeslice}. |
|  | #' @param classProbs a logical; should class probabilities be computed for |
|  | #' classification models (along with predicted values) in each resample? |
|  | #' @param summaryFunction a function to compute performance metrics across |
|  | #' resamples. The arguments to the function should be the same as those in |
|  | #' \code{\link{defaultSummary}}. Note that if \code{method = "oob"} is used, |
|  | #' this option is ignored and a warning is issued. |
|  | #' @param selectionFunction the function used to select the optimal tuning |
|  | #' parameter. This can be a name of the function or the function itself. See |
|  | #' \code{\link{best}} for details and other options. |
|  | #' @param preProcOptions A list of options to pass to \code{\link{preProcess}}. |
|  | #' The type of pre-processing (e.g. center, scaling etc) is passed in via the |
|  | #' \code{preProc} option in \code{\link{train}}. |
|  | #' @param sampling a single character value describing the type of additional |
|  | #' sampling that is conducted after resampling (usually to resolve class |
|  | #' imbalances). Values are \code{"none"}, \code{"down"}, \code{"up"}, |
|  | #' \code{"smote"}, or \code{"rose"}. The latter two values require the |
|  | #' \pkg{DMwR} and \pkg{ROSE} packages, respectively. This argument can also be |
|  | #' a list to facilitate custom sampling and these details can be found on the |
|  | #' \pkg{caret} package website for sampling (link below). |
|  | #' @param index a list with elements for each resampling iteration. Each list |
|  | #' element is a vector of integers corresponding to the rows used for training |
|  | #' at that iteration. |
|  | #' @param indexOut a list (the same length as \code{index}) that dictates which |
|  | #' data are held-out for each resample (as integers). If \code{NULL}, then the |
|  | #' unique set of samples not contained in \code{index} is used. |
|  | #' @param indexFinal an optional vector of integers indicating which samples |
|  | #' are used to fit the final model after resampling. If \code{NULL}, then |
|  | #' entire data set is used. |
|  | #' @param timingSamps the number of training set samples that will be used to |
|  | #' measure the time for predicting samples (zero indicates that the prediction |
|  | #' time should not be estimated. |
|  | #' @param predictionBounds a logical or numeric vector of length 2 (regression |
|  | #' only). If logical, the predictions can be constrained to be within the limit |
|  | #' of the training set outcomes. For example, a value of \code{c(TRUE, FALSE)} |
|  | #' would only constrain the lower end of predictions. If numeric, specific |
|  | #' bounds can be used. For example, if \code{c(10, NA)}, values below 10 would |
|  | #' be predicted as 10 (with no constraint in the upper side). |
|  | #' @param seeds an optional set of integers that will be used to set the seed |
|  | #' at each resampling iteration. This is useful when the models are run in |
|  | #' parallel. A value of \code{NA} will stop the seed from being set within the |
|  | #' worker processes while a value of \code{NULL} will set the seeds using a |
|  | #' random set of integers. Alternatively, a list can be used. The list should |
|  | #' have \code{B+1} elements where \code{B} is the number of resamples, unless |
|  | #' \code{method} is \code{"boot632"} in which case \code{B} is the number of |
|  | #' resamples plus 1. The first \code{B} elements of the list should be vectors |
|  | #' of integers of length \code{M} where \code{M} is the number of models being |
|  | #' evaluated. The last element of the list only needs to be a single integer |
|  | #' (for the final model). See the Examples section below and the Details |
|  | #' section. |
|  | #' @param adaptive a list used when \code{method} is \code{"adaptive\_cv"}, |
|  | #' \code{"adaptive\_boot"} or \code{"adaptive\_LGOCV"}. See Details below. |
|  | #' @param trim a logical. If \code{TRUE} the final model in |
|  | #' \code{object\$finalModel} may have some components of the object removed so |
|  | #' reduce the size of the saved object. The \code{predict} method will still |
|  | #' work, but some other features of the model may not work. \code{trim}ing will |
|  | #' occur only for models where this feature has been implemented. |
|  | #' @param allowParallel if a parallel backend is loaded and available, should |
|  | #' the function use it? |
|  | #' @return An echo of the parameters specified |
|  | #' @author Max Kuhn |
|  | #' @references Efron (1983). ``Estimating the error rate of a prediction rule: |
|  | #' improvement on cross-validation''. Journal of the American Statistical |
|  | #' Association, 78(382):316-331 |
|  | #' |
|  | #' Efron, B., & Tibshirani, R. J. (1994). ``An introduction to the bootstrap'', |
|  | #' pages 249-252. CRC press. |
|  | #' |
|  | #' Bergstra and Bengio (2012), ``Random Search for Hyper-Parameter |
|  | #' Optimization'', Journal of Machine Learning Research, 13(Feb):281-305 |
|  | #' |
|  | #' Kuhn (2014), ``Futility Analysis in the Cross-Validation of Machine Learning |
|  | #' Models'' \url{http://arxiv.org/abs/1405.6974}, |
|  | #' |
|  | #' Package website for subsampling: |
|  | #' \url{https://topepo.github.io/caret/subsampling-for-class-imbalances.html} |
|  | #' @keywords utilities |
|  | #' @examples |
|  | #' |
|  | #' \dontrun{ |
|  | #' |
|  | #' ## Do 5 repeats of 10-Fold CV for the iris data. We will fit |
|  | #' ## a KNN model that evaluates 12 values of k and set the seed |
|  | #' ## at each iteration. |
|  | #' |
|  | #' set.seed(123) |
|  | #' seeds <- vector(mode = "list", length = 51) |
|  | #' for(i in 1:50) seeds[[i]] <- sample.int(1000, 22) |
|  | #' |
|  | #' ## For the last model: |
|  | #' seeds[[51]] <- sample.int(1000, 1) |
|  | #' |
|  | #' ctrl <- trainControl(method = "repeatedcv", |
|  | #' repeats = 5, |
|  | #' seeds = seeds) |
|  | #' |
|  | #' set.seed(1) |
|  | #' mod <- train(Species ~ ., data = iris, |
|  | #' method = "knn", |
|  | #' tuneLength = 12, |
|  | #' trControl = ctrl) |
|  | #' |
|  | #' |
|  | #' ctrl2 <- trainControl(method = "adaptive\_cv", |
|  | #' repeats = 5, |
|  | #' verboseIter = TRUE, |
|  | #' seeds = seeds) |
|  | #' |
|  | #' set.seed(1) |
|  | #' mod2 <- train(Species ~ ., data = iris, |
|  | #' method = "knn", |
|  | #' tuneLength = 12, |
|  | #' trControl = ctrl2) |
|  | #' |
|  | #' } |
|  | #' |
|  | #' @export trainControl |
|  | trainControl <- function(method = "boot", |
|  | number = ifelse(grepl("cv", method), 10, 25), |
|  | repeats = ifelse(grepl("[d\_]cv$", method), 1, NA), |
|  | p = .75, |
|  | search = "grid", |
|  | initialWindow = NULL, |
|  | horizon = 1, |
|  | fixedWindow = TRUE, |
|  | skip = 0, |
|  | verboseIter = FALSE, |
|  | returnData = TRUE, |
|  | returnResamp = "final", |
|  | savePredictions = FALSE, |
|  | classProbs = FALSE, |
|  | summaryFunction = defaultSummary, |
|  | selectionFunction = "best", |
|  | preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5, |
|  | freqCut = 95/5, uniqueCut = 10, |
|  | cutoff = 0.9), |
|  | sampling = NULL, |
|  | index = NULL, |
|  | indexOut = NULL, |
|  | indexFinal = NULL, |
|  | timingSamps = 0, |
|  | predictionBounds = rep(FALSE, 2), |
|  | seeds = NA, |
|  | adaptive = list(min = 5, alpha = 0.05, method = "gls", complete = TRUE), |
|  | trim = FALSE, |
|  | allowParallel = TRUE) |
|  | { |
|  | if(is.null(selectionFunction)) stop("null selectionFunction values not allowed") |
|  | if(!(returnResamp %in% c("all", "final", "none"))) stop("incorrect value of returnResamp") |
|  | if(length(predictionBounds) > 0 && length(predictionBounds) != 2) stop("'predictionBounds' should be a logical or numeric vector of length 2") |
|  | if(any(names(preProcOptions) == "method")) stop("'method' cannot be specified here") |
|  | if(any(names(preProcOptions) == "x")) stop("'x' cannot be specified here") |
|  | if(!is.na(repeats) & !(method %in% c("repeatedcv", "adaptive\_cv"))) |
|  | warning("`repeats` has no meaning for this resampling method.", call. = FALSE) |
|  |  |
|  | if(!(adaptive$method %in% c("gls", "BT"))) stop("incorrect value of adaptive$method") |
|  | if(adaptive$alpha < .0000001 | adaptive$alpha > 1) stop("incorrect value of adaptive$alpha") |
|  | if(grepl("adapt", method)) { |
|  | num <- if(method == "adaptive\_cv") number\*repeats else number |
|  | if(adaptive$min >= num) stop(paste("adaptive$min should be less than", num)) |
|  | if(adaptive$min <= 1) stop("adaptive$min should be greater than 1") |
|  | } |
|  | if(!(search %in% c("grid", "random"))) |
|  | stop("`search` should be either 'grid' or 'random'") |
|  | if(method == "oob" & any(names(match.call()) == "summaryFunction")) { |
|  | warning("Custom summary measures cannot be computed for out-of-baf resampling. ", |
|  | "This value of `summaryFunction` will be ignored.", |
|  | call. = FALSE) |
|  | } |
|  |  |
|  | list(method = method, |
|  | number = number, |
|  | repeats = repeats, |
|  | search = search, |
|  | p = p, |
|  | initialWindow = initialWindow, |
|  | horizon = horizon, |
|  | fixedWindow = fixedWindow, |
|  | skip = skip, |
|  | verboseIter = verboseIter, |
|  | returnData = returnData, |
|  | returnResamp = returnResamp, |
|  | savePredictions = savePredictions, |
|  | classProbs = classProbs, |
|  | summaryFunction = summaryFunction, |
|  | selectionFunction = selectionFunction, |
|  | preProcOptions = preProcOptions, |
|  | sampling = sampling, |
|  | index = index, |
|  | indexOut = indexOut, |
|  | indexFinal = indexFinal, |
|  | timingSamps = timingSamps, |
|  | predictionBounds = predictionBounds, |
|  | seeds = seeds, |
|  | adaptive = adaptive, |
|  | trim = trim, |
|  | allowParallel = allowParallel) |
|  | } |