Package: mlearning (via r-universe)

July 26, 2024

Type Package

Version 1.2.1

Title Machine Learning Algorithms with Unified Interface and Confusion Matrices

Description A unified interface is provided to various machine learning algorithms like linear or quadratic discriminant analysis, k-nearest neighbors, random forest, support vector machine, ... It allows to train, test, and apply cross-validation using similar functions and function arguments with a minimalist and clean, formula-based interface. Missing data are processed the same way as base and stats R functions for all algorithms, both in training and testing. Confusion matrices are also provided with a rich set of metrics calculated and a few specific plots.

Maintainer Philippe Grosjean <phgrosjean@sciviews.org>

Depends R (>= 3.0.4)

Imports stats, grDevices, class, nnet, MASS, e1071, randomForest, ipred, rpart

Suggests mlbench, datasets, RColorBrewer, spelling, knitr, rmarkdown, covr

URL https://www.sciviews.org/mlearning/

BugReports https://github.com/SciViews/mlearning/issues

License GPL (>= 2)

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Config/testthat/edition 3

Encoding UTF-8

Language en-US

Repository https://sciviews.r-universe.dev

RemoteUrl https://github.com/SciViews/mlearning

RemoteRef HEAD

RemoteSha 2ff68b8445b5538b2e4362b2f505c68ced6caa1e

Contents

`mlearning-package` *Machine Learning Algorithms with Unified Interface and Confusion Matrices*

Description

This package provides wrappers around several existing machine learning algorithms in R, under a unified user interface. Confusion matrices can also be calculated and viewed as tables or plots. Key features are:

- Unified, formula-based interface for all algorithms, similar to `stats::lm()`.
- Optimized code when a simplified formula `y ~ .` is used, meaning all variables in data are used (one of them (`y` here) is the class to be predicted (classification problem, a factor variable), or the dependent variable of the model (regression problem, a numeric variable).
- Similar way of dealing with missing data, both in the training set and in predictions. Underlying algorithms deal differently with missing data. Some accept them, other not.
- Unified way of dealing with factor levels that have no cases in the training set. The training succeeds, but the classifier is, of course, unable to classify items in the missing class.
- The `predict()` methods have similar arguments. They return the class, membership to the classes, both, or something else (probabilities, raw predictions, ...) depending on the algorithm or the problem (classification or regression).

- The `cvpredict()` method is available for all algorithms and it performs very easily a cross-validation, or even a leave_one_out validation (when `cv.k` = number of cases). It operates transparently for the end-user.
- The `confusion()` method creates a confusion matrix and the object can be printed, summarized, plotted. Various metrics are easily derived from the confusion matrix. Also, it allows to adjust prior probabilities of the classes in a classification problem, in order to obtain more representative estimates of the metrics when priors are adjusted to values closes to real proportions of classes in the data.

See `mlearning()` for further explanations and an example analysis. See `mlLda()` for examples of the different forms of the formula that can be used. See `plot.confusion()` for the different ways to explore the confusion matrix.

Important functions

- `ml_lda()`, `ml_qda()`, `ml_naive_bayes()`, `ml_knn()`, `ml_lvq()`, `ml_nnet()`, `ml_rpart()`, `ml_rforest()` and `ml_svm()` to train classifiers or regressors with the different algorithms that are supported in the package,
- `predict()` and `cvpredict()` for predictions, including using cross-validation,
- `confusion()` to calculate the confusion matrix (with various methods to analyze it and to calculate derived metrics like recall, precision, F-score, ...)
- `prior()` to adjust prior probabilities,
- `response()` and `train()` to extract response and training variables from an **mlearning** object.

`confusion` *Construct and analyze confusion matrices*

Description

Confusion matrices compare two classifications (usually one done automatically using a machine learning algorithm versus the true classification done by a specialist... but one can also compare two automatic or two manual classifications against each other).

Usage

```
confusion(x, ...)

## Default S3 method:
confusion(
  x,
  y = NULL,
  vars = c("Actual", "Predicted"),
  labels = vars,
  merge.by = "Id",
  useNA = "ifany",
  prior,
```

```
  ...
)

## S3 method for class 'mlearning'
confusion(
  x,
  y = response(x),
  labels = c("Actual", "Predicted"),
  useNA = "ifany",
  prior,
  ...
)

## S3 method for class 'confusion'
print(x, sums = TRUE, error.col = sums, digits = 0, sort = "ward.D2", ...)

## S3 method for class 'confusion'
summary(object, type = "all", sort.by = "Fscore", decreasing = TRUE, ...)

## S3 method for class 'summary.confusion'
print(x, ...)
```

Arguments

x	an object with a confusion() method implemented.
...	further arguments passed to the method.
y	another object, from which to extract the second classification, or NULL if not used.
vars	the variables of interest in the first and second classification in the case the objects are lists or data frames. Otherwise, this argument is ignored and x and y must be factors with same length and same levels.
labels	labels to use for the two classifications. By default, they are the same as vars, or the one in the confusion matrix.
merge.by	a character string with the name of variables to use to merge the two data frames, or NULL.
useNA	do we keep NAs as a separate category? The default "ifany" creates this category only if there are missing values. Other possibilities are "no", or "always".
prior	class frequencies to use for first classifier that is tabulated in the rows of the confusion matrix. For its value, see here under, the value= argument.
sums	is the confusion matrix printed with rows and columns sums?
error.col	is a column with class error for first classifier added (equivalent to false negative rate of FNR)?
digits	the number of digits after the decimal point to print in the confusion matrix. The default or zero leads to most compact presentation and is suitable for frequencies, but not for relative frequencies.

sort	are rows and columns of the confusion matrix sorted so that classes with larger confusion are closer together? Sorting is done using a hierarchical clustering with hclust(). The clustering method is "ward.D2" by default, but see the hclust() help for other options). If FALSE or NULL, no sorting is done.
object	a **confusion** object
type	either "all" (by default), or considering TP is the true positives, FP is the false positives, TN is the true negatives and FN is the false negatives, one can also specify: "Fscore" (F-score = F-measure = F1 score = harmonic mean of Precision and recall), "Recall" (TP / (TP + FN) = 1 - FNR), "Precision" (TP / (TP + FP) = 1 - FDR), "Specificity" (TN / (TN + FP) = 1 - FPR), "NPV" (Negative predicted value = TN / (TN + FN) = 1 - FOR), "FPR" (False positive rate = 1 - Specificity = FP / (FP + TN)), "FNR" (False negative rate = 1 - Recall = FN / (TP + FN)), "FDR" (False Discovery Rate = 1 - Precision = FP / (TP + FP)), "FOR" (False omission rate = 1 - NPV = FN / (FN + TN)), "LRPT" (Likelihood Ratio for Positive Tests = Recall / FPR = Recall / (1 - Specificity)), "LRNT" Likelihood Ratio for Negative Tests = FNR / Specificity = (1 - Recall) / Specificity, "LRPS" (Likelihood Ratio for Positive Subjects = Precision / FOR = Precision / (1 - NPV)), "LRNS" (Likelihood Ratio Negative Subjects = FDR / NPV = (1 - Precision) / (1 - FOR)), "BalAcc" (Balanced accuracy = (Sensitivity + Specificity) / 2), "MCC" (Matthews correlation coefficient), "Chisq" (Chisq metric), or "Bray" (Bray-Curtis metric)
sort.by	the statistics to use to sort the table (by default, Fmeasure, the F1 score for each class = 2 * recall * precision / (recall + precision)).
decreasing	do we sort in increasing or decreasing order?

Value

A confusion matrix in a **confusion** object.

See Also

mlearning(), plot.confusion(), prior()

Examples

```
data("Glass", package = "mlbench")
# Use a little bit more informative labels for Type
Glass$Type <- as.factor(paste("Glass", Glass$Type))

# Use learning vector quantization to classify the glass types
# (using default parameters)
summary(glass_lvq <- ml_lvq(Type ~ ., data = Glass))

# Calculate cross-validated confusion matrix
(glass_conf <- confusion(cvpredict(glass_lvq), Glass$Type))
# Raw confusion matrix: no sort and no margins
print(glass_conf, sums = FALSE, sort = FALSE)

summary(glass_conf)
summary(glass_conf, type = "Fscore")
```

`mlearning` *Machine learning model for (un)supervised classification or regression*

Description

An **mlearning** object provides an unified (formula-based) interface to several machine learning algorithms. They share the same interface and very similar arguments. They conform to the formula-based approach, of say, `stats::lm()` in base R, but with a coherent handling of missing data and missing class levels. An optimized version exists for the simplified `y ~ .` formula. Finally, cross-validation is also built-in.

Usage

```
mlearning(
  formula,
  data,
  method,
  model.args,
  call = match.call(),
  ...,
  subset,
  na.action = na.fail
)

## S3 method for class 'mlearning'
print(x, ...)

## S3 method for class 'mlearning'
summary(object, ...)

## S3 method for class 'summary.mlearning'
print(x, ...)

## S3 method for class 'mlearning'
plot(x, y, ...)

## S3 method for class 'mlearning'
predict(
  object,
  newdata,
  type = c("class", "membership", "both"),
  method = c("direct", "cv"),
  na.action = na.exclude,
  ...
)
```

```
cvpredict(object, ...)

## S3 method for class 'mlearning'
cvpredict(
  object,
  type = c("class", "membership", "both"),
  cv.k = 10,
  cv.strat = TRUE,
  ...
)
```

Arguments

formula	a formula with left term being the factor variable to predict (for supervised classification), a vector of numbers (for regression) or nothing (for unsupervised classification) and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the `class ~ .` short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using `I()`). Supervised classification, regression or unsupervised classification are not available for all algorithms. Check respective help pages.
data	a data.frame to use as a training set.
method	`"direct"` (default) or `"cv"`. `"direct"` predicts new cases in `newdata=` if this argument is provided, or the cases in the training set if not. Take care that not providing `newdata=` means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different dataset in `newdata=` or use the alternate cross-validation ("cv") technique. If you specify `method = "cv"` then `cvpredict()` is used and you cannot provide `newdata=` in that case. Other methods may be provided by the various algorithms (check their help pages)
model.args	arguments for formula modeling with substituted data and subset... Not to be used by the end-user.
call	the function call. Not to be used by the end-user.
...	further arguments (depends on the method).
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).
na.action	function to specify the action to be taken if `NA`s are found. For `ml_qda()` `na.fail` is used by default. The calculation is stopped if there is any `NA` in the data. Another option is `na.omit`, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the `predict()` method, the default, and most suitable option, is `na.exclude`. In that case, rows with `NA`s in `newdata=` are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as `newdata=`).

x, object	an **mlearning** object
y	a second **mlearning** object or nothing (not used in several plots)
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. "class" by default, the predicted classes. Other options are "membership" the membership (a number between 0 and 1) to the different classes, or "both" to return classes and memberships. Other types may be provided for some algorithms (read respective help pages).
cv.k	k for k-fold cross-validation, cf ipred::errorest(). By default, 10.
cv.strat	is the subsampling stratified or not in cross-validation, cf ipred::errorest(). TRUE by default.

Value

an **mlearning** object for mlearning(). Methods return their own results that can be a **mlearning**, **data.frame**, **vector**, etc.

See Also

ml_lda(), ml_qda(), ml_naive_bayes(), ml_nnet(), ml_rpart(), ml_rforest(), ml_svm(), confusion() and prior(). Also ipred::errorest() that internally computes the cross-validation in cvpredict().

Examples

```
# mlearning() should not be calle directly. Use the mlXXX() functions instead
# for instance, for Random Forest, use ml_rforest()/mlRforest()
# A typical classification involves several steps:
#
# 1) Prepare data: split into training set (2/3) and test set (1/3)
#    Data cleaning (elimination of unwanted variables), transformation of
#    others (scaling, log, ratios, numeric to factor, ...) may be necessary
#    here. Apply the same treatments on the training and test sets
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133) # Also random or stratified sampling
iris_train <- iris[train, ]
iris_test <- iris[-train, ]

# 2) Train the classifier, use of the simplified formula class ~ . encouraged
#    so, you may have to prepare the train/test sets to keep only relevant
#    variables and to possibly transform them before use
iris_rf <- ml_rforest(data = iris_train, Species ~ .)
iris_rf
summary(iris_rf)
train(iris_rf)
response(iris_rf)

# 3) Find optimal values for the parameters of the model
#    This is usally done iteratively. Just an example with ntree where a plot
```

```
#    exists to help finding optimal value
plot(iris_rf)
# For such a relatively simple case, 50 trees are enough, retrain with it
iris_rf <- ml_rforest(data = iris_train, Species ~ ., ntree = 50)
summary(iris_rf)

# 4) Study the classifier performances. Several metrics and tools exists
#    like ROC curves, AUC, etc. Tools provided here are the confusion matrix
#    and the metrics that are calculated on it.
predict(iris_rf) # Default type is class
predict(iris_rf, type = "membership")
predict(iris_rf, type = "both")
# Confusion matrice and metrics using 10-fols cross-validation
iris_rf_conf <- confusion(iris_rf, method = "cv")
iris_rf_conf
summary(iris_rf_conf)
# Note you may want to manipulate priors too, see ?prior

# 5) Go back to step #1 and refine the process until you are happy with the
#    results. Then, you can use the classifier to predict unknown items.
```

`mlKnn` *Supervised classification using k-nearest neighbor*

Description

Unified (formula-based) interface version of the k-nearest neighbor algorithm provided by `class::knn()`.

Usage

```
mlKnn(train, ...)

ml_knn(train, ...)

## S3 method for class 'formula'
mlKnn(formula, data, k.nn = 5, ..., subset, na.action)

## Default S3 method:
mlKnn(train, response, k.nn = 5, ...)

## S3 method for class 'mlKnn'
summary(object, ...)

## S3 method for class 'summary.mlKnn'
print(x, ...)

## S3 method for class 'mlKnn'
predict(
  object,
```

```
  newdata,
  type = c("class", "prob", "both"),
  method = c("direct", "cv"),
  na.action = na.exclude,
  ...
)
```

Arguments

train	a matrix or data frame with predictors.
...	further arguments passed to the classification method or its predict() method (not used here for now).
formula	a formula with left term being the factor variable to predict and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the class ~ . short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using I()).
data	a data.frame to use as a training set.
k.nn	k used for k-NN number of neighbor considered. Default is 5.
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).
na.action	function to specify the action to be taken if NAs are found. For ml_knn() na.fail is used by default. The calculation is stopped if there is any NA in the data. Another option is na.omit, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the predict() method, the default, and most suitable option, is na.exclude. In that case, rows with NAs in newdata= are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as newdata=).
response	a vector of factor for the classification.
x, object	an **mlKnn** object
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. "class" by default, the predicted classes. Other options are "prob" the "probability" for the different classes as assessed by the number of neighbors of these classes, or "both" to return classes and "probabilities",
method	"direct" (default) or "cv". "direct" predicts new cases in newdata= if this argument is provided, or the cases in the training set if not. Take care that not providing newdata= means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different data set in newdata= or use the alternate cross-validation ("cv") technique. If you specify method = "cv" then cvpredict() is used and you cannot provide newdata= in that case.

Value

ml_knn()/mlKnn() creates an **mlKnn**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also class::knn() and ipred::predict.ipredknn() that actually do the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_knn <- ml_knn(data = iris_train, Species ~ .)
summary(iris_knn)
predict(iris_knn) # This object only returns classes
# Self-consistency, do not use for assessing classifier performances!
confusion(iris_knn)
# Use an independent test set instead
confusion(predict(iris_knn, newdata = iris_test), iris_test$Species)
```

---

mlLda *Supervised classification using linear discriminant analysis*

Description

Unified (formula-based) interface version of the linear discriminant analysis algorithm provided by MASS::lda().

Usage

```
mlLda(train, ...)

ml_lda(train, ...)

## S3 method for class 'formula'
mlLda(formula, data, ..., subset, na.action)

## Default S3 method:
```

```
mlLda(train, response, ...)

## S3 method for class 'mlLda'
predict(
  object,
  newdata,
  type = c("class", "membership", "both", "projection"),
  prior = object$prior,
  dimension = NULL,
  method = c("plug-in", "predictive", "debiased", "cv"),
  ...
)
```

Arguments

train	a matrix or data frame with predictors.
...	further arguments passed to MASS::lda() or its predict() method (see the corresponding help page).
formula	a formula with left term being the factor variable to predict and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the class ~ . short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using I()).
data	a data.frame to use as a training set.
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).
na.action	function to specify the action to be taken if NAs are found. For ml_lda() na.fail is used by default. The calculation is stopped if there is any NA in the data. Another option is na.omit, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the predict() method, the default, and most suitable option, is na.exclude. In that case, rows with NAs in newdata= are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as newdata=).
response	a vector of factor for the classification.
object	an **mlLda** object
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. "class" by default, the predicted classes. Other options are "membership" the membership (a number between 0 and 1) to the different classes, or "both" to return classes and memberships. The type = "projection" returns a projection of the individuals in the plane represented by the dimension= discriminant components.

prior — the prior probabilities of class membership. By default, the prior are obtained from the object and, if they where not changed, correspond to the proportions observed in the training set.

dimension — the number of the predictive space to use. If NULL (the default) a reasonable value is used. If this is less than min(p, ng-1), only the first dimension discriminant components are used (except for method = "predictive"), and only those dimensions are returned in x.

method — "plug-in", "predictive", "debiased", or "cv". "plug-in" (default) the usual unbiased parameter estimates are used. With "predictive", the parameters are integrated out using a vague prior. With "debiased", an unbiased estimator of the log posterior probabilities is used. With "cv", cross-validation is used instead. If you specify method = "cv" then cvpredict() is used and you cannot provide newdata= in that case.

Value

ml_lda()/mlLda() creates an **mlLda**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also MASS::lda() that actually does the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_lda <- ml_lda(data = iris_train, Species ~ .)
iris_lda
summary(iris_lda)
plot(iris_lda, col = as.numeric(response(iris_lda)) + 1)
# Prediction using a test set
predict(iris_lda, newdata = iris_test) # class (default type)
predict(iris_lda, type = "membership") # posterior probability
predict(iris_lda, type = "both") # both class and membership in a list
# Type projection
predict(iris_lda, type = "projection") # Projection on the LD axes
# Add test set items to the previous plot
points(predict(iris_lda, newdata = iris_test, type = "projection"),
  col = as.numeric(predict(iris_lda, newdata = iris_test)) + 1, pch = 19)
# predict() and confusion() should be used on a separate test set
# for unbiased estimation (or using cross-validation, bootstrap, ...)
```

```
# Wrong, cf. biased estimation (so-called, self-consistency)
confusion(iris_lda)
# Estimation using a separate test set
confusion(predict(iris_lda, newdata = iris_test), iris_test$Species)

# Another dataset (binary predictor... not optimal for lda, just for test)
data("HouseVotes84", package = "mlbench")
house_lda <- ml_lda(data = HouseVotes84, na.action = na.omit, Class ~ .)
summary(house_lda)
confusion(house_lda) # Self-consistency (biased metrics)
print(confusion(house_lda), error.col = FALSE) # Without error column

# More complex formulas
# Exclude one or more variables
iris_lda2 <- ml_lda(data = iris, Species ~ . - Sepal.Width)
summary(iris_lda2)
# With calculation
iris_lda3 <- ml_lda(data = iris, Species ~ log(Petal.Length) +
  log(Petal.Width) + I(Petal.Length/Sepal.Length))
summary(iris_lda3)

# Factor levels with missing items are allowed
ir2 <- iris[-(51:100), ] # No Iris versicolor in the training set
iris_lda4 <- ml_lda(data = ir2, Species ~ .)
summary(iris_lda4) # missing class
# Missing levels are reinjected in class or membership by predict()
predict(iris_lda4, type = "both")
# ... but, of course, the classifier is wrong for Iris versicolor
confusion(predict(iris_lda4, newdata = iris), iris$Species)

# Simpler interface, but more memory-effective
iris_lda5 <- ml_lda(train = iris[, -5], response = iris$Species)
summary(iris_lda5)
```

`mlLvq` *Supervised classification using learning vector quantization*

Description

Unified (formula-based) interface version of the learning vector quantization algorithms provided by `class::olvq1()`, `class::lvq1()`, `class::lvq2()`, and `class::lvq3()`.

Usage

```
mlLvq(train, ...)

ml_lvq(train, ...)

## S3 method for class 'formula'
mlLvq(
```

```
  formula,
  data,
  k.nn = 5,
  size,
  prior,
  algorithm = "olvq1",
  ...,
  subset,
  na.action
)

## Default S3 method:
mlLvq(train, response, k.nn = 5, size, prior, algorithm = "olvq1", ...)

## S3 method for class 'mlLvq'
summary(object, ...)

## S3 method for class 'summary.mlLvq'
print(x, ...)

## S3 method for class 'mlLvq'
predict(
  object,
  newdata,
  type = "class",
  method = c("direct", "cv"),
  na.action = na.exclude,
  ...
)
```

Arguments

train	a matrix or data frame with predictors.
...	further arguments passed to the classification method or its predict() method (not used here for now).
formula	a formula with left term being the factor variable to predict and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the class ~ . short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using I()).
data	a data.frame to use as a training set.
k.nn	k used for k-NN number of neighbor considered. Default is 5.
size	the size of the codebook. Defaults to min(round(0.4 * nc * (nc - 1 + p/2),0), n) where nc is the number of classes.
prior	probabilities to represent classes in the codebook (default values are the proportions in the training set).

algorithm	"olvq1" (by default, the optimized 'lvq1' version), or "lvq1", "lvq2", "lvq3".
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).
na.action	function to specify the action to be taken if NAs are found. For [ml_lvq)] na.fail is used by default. The calculation is stopped if there is any NA in the data. Another option is na.omit, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the predict() method, the default, and most suitable option, is na.exclude. In that case, rows with NAs in newdata= are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as newdata=). [ml_lvq)]: R:ml_lvq)
response	a vector of factor of the classes.
x, object	an **mlLvq** object
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. For this method, only "class" is accepted, and it is the default. It returns the predicted classes.
method	"direct" (default) or "cv". "direct" predicts new cases in newdata= if this argument is provided, or the cases in the training set if not. Take care that not providing newdata= means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different dataset in newdata= or use the alternate cross-validation ("cv") technique. If you specify method = "cv" then cvpredict() is used and you cannot provide newdata= in that case.

Value

ml_lvq()/mlLvq() creates an **mlLvq**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also class::olvq1(), class::lvq1(), class::lvq2(), and class::lvq3() that actually do the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
```

```
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_lvq <- ml_lvq(data = iris_train, Species ~ .)
summary(iris_lvq)
predict(iris_lvq) # This object only returns classes
#' # Self-consistency, do not use for assessing classifier performances!
confusion(iris_lvq)
# Use an independent test set instead
confusion(predict(iris_lvq, newdata = iris_test), iris_test$Species)
```

`mlNaiveBayes` *Supervised classification using naive Bayes*

Description

Unified (formula-based) interface version of the naive Bayes algorithm provided by `e1071::naiveBayes()`.

Usage

```
mlNaiveBayes(train, ...)

ml_naive_bayes(train, ...)

## S3 method for class 'formula'
mlNaiveBayes(formula, data, laplace = 0, ..., subset, na.action)

## Default S3 method:
mlNaiveBayes(train, response, laplace = 0, ...)

## S3 method for class 'mlNaiveBayes'
predict(
  object,
  newdata,
  type = c("class", "membership", "both"),
  method = c("direct", "cv"),
  na.action = na.exclude,
  threshold = 0.001,
  eps = 0,
  ...
)
```

Arguments

`train`	a matrix or data frame with predictors.
`...`	further arguments passed to the classification method or its `predict()` method (not used here for now).

formula	a formula with left term being the factor variable to predict and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the `class ~ .` short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using `I()`).
data	a data.frame to use as a training set.
laplace	positive number controlling Laplace smoothing for the naive Bayes classifier. The default (0) disables Laplace smoothing.
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).
na.action	function to specify the action to be taken if NAs are found. For `ml_naive_bayes()` `na.fail` is used by default. The calculation is stopped if there is any NA in the data. Another option is `na.omit`, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the `predict()` method, the default, and most suitable option, is `na.exclude`. In that case, rows with NAs in `newdata=` are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as `newdata=`).
response	a vector of factor with the classes.
object	an **mlNaiveBayes** object
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. `"class"` by default, the predicted classes. Other options are `"membership"`, the posterior probability or `"both"` to return classes and memberships,
method	`"direct"` (default) or `"cv"`. `"direct"` predicts new cases in `newdata=` if this argument is provided, or the cases in the training set if not. Take care that not providing `newdata=` means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different dataset in `newdata=` or use the alternate cross-validation ("cv") technique. If you specify `method = "cv"` then `cvpredict()` is used and you cannot provide `newdata=` in that case.
threshold	value replacing cells with probabilities within 'eps' range.
eps	number for specifying an epsilon-range to apply Laplace smoothing (to replace zero or close-zero probabilities by 'threshold').

Value

`ml_naive_bayes()`/`mlNaiveBayes()` creates an **mlNaiveBayes**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like `predict()` or `cvpredict()`. In case you want to program new functions or extract specific components, inspect the "unclassed" object using `unclass()`.

See Also

mlearning(), cvpredict(), confusion(), also e1071::naiveBayes() that actually does the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_nb <- ml_naive_bayes(data = iris_train, Species ~ .)
summary(iris_nb)
predict(iris_nb) # Default type is class
predict(iris_nb, type = "membership")
predict(iris_nb, type = "both")
# Self-consistency, do not use for assessing classifier performances!
confusion(iris_nb)
# Use an independent test set instead
confusion(predict(iris_nb, newdata = iris_test), iris_test$Species)

# Another dataset
data("HouseVotes84", package = "mlbench")
house_nb <- ml_naive_bayes(data = HouseVotes84, Class ~ .,
  na.action = na.omit)
summary(house_nb)
confusion(house_nb) # Self-consistency
confusion(cvpredict(house_nb), na.omit(HouseVotes84)$Class)
```

---

mlNnet *Supervised classification and regression using neural network*

Description

Unified (formula-based) interface version of the single-hidden-layer neural network algorithm, possibly with skip-layer connections provided by nnet::nnet().

Usage

```
mlNnet(train, ...)

ml_nnet(train, ...)

## S3 method for class 'formula'
mlNnet(
```

```
  formula,
  data,
  size = NULL,
  rang = NULL,
  decay = 0,
  maxit = 1000,
  ...,
  subset,
  na.action
)

## Default S3 method:
mlNnet(train, response, size = NULL, rang = NULL, decay = 0, maxit = 1000, ...)

## S3 method for class 'mlNnet'
predict(
  object,
  newdata,
  type = c("class", "membership", "both", "raw"),
  method = c("direct", "cv"),
  na.action = na.exclude,
  ...
)
```

Arguments

train	a matrix or data frame with predictors.
...	further arguments passed to nnet::nnet() that has many more parameters (see its help page).
formula	a formula with left term being the factor variable to predict (for supervised classification), a vector of numbers (for regression) and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the class ~ . short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using I()).
data	a data.frame to use as a training set.
size	number of units in the hidden layer. Can be zero if there are skip-layer units. If NULL (the default), a reasonable value is computed.
rang	initial random weights on [-rang, rang]. Value about 0.5 unless the inputs are large, in which case it should be chosen so that rang * max(|x|) is about 1. If NULL, a reasonable default is computed.
decay	parameter for weight decay. Default to 0.
maxit	maximum number of iterations. Default 1000 (it is 100 in nnet::nnet()).
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).

na.action	function to specify the action to be taken if NAs are found. For ml_nnet() na.fail is used by default. The calculation is stopped if there is any NA in the data. Another option is na.omit, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the predict() method, the default, and most suitable option, is na.exclude. In that case, rows with NAs in newdata= are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as newdata=).
response	a vector of factor (classification) or numeric (regression).
object	an **mlNnet** object
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. "class" by default, the predicted classes. Other options are "membership" the membership (number between 0 and 1) to the different classes, or "both" to return classes and memberships. Also type "raw" as non normalized result as returned by nnet::nnet() (useful for regression, see examples).
method	"direct" (default) or "cv". "direct" predicts new cases in newdata= if this argument is provided, or the cases in the training set if not. Take care that not providing newdata= means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different data set in newdata= or use the alternate cross-validation ("cv") technique. If you specify method = "cv" then cvpredict() is used and you cannot provide newdata= in that case.

Value

ml_nnet()/mlNnet() creates an **mlNnet**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also nnet::nnet() that actually does the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA
```

```
set.seed(689) # Useful for reproductibility, use a different value each time!
iris_nnet <- ml_nnet(data = iris_train, Species ~ .)
summary(iris_nnet)
predict(iris_nnet) # Default type is class
predict(iris_nnet, type = "membership")
predict(iris_nnet, type = "both")
# Self-consistency, do not use for assessing classifier performances!
confusion(iris_nnet)
# Use an independent test set instead
confusion(predict(iris_nnet, newdata = iris_test), iris_test$Species)

# Idem, but two classes prediction
data("HouseVotes84", package = "mlbench")
set.seed(325)
house_nnet <- ml_nnet(data = HouseVotes84, Class ~ ., na.action = na.omit)
summary(house_nnet)
# Cross-validated confusion matrix
confusion(cvpredict(house_nnet), na.omit(HouseVotes84)$Class)

# Regression
data(airquality, package = "datasets")
set.seed(74)
ozone_nnet <- ml_nnet(data = airquality, Ozone ~ ., na.action = na.omit,
  skip = TRUE, decay = 1e-3, size = 20, linout = TRUE)
summary(ozone_nnet)
plot(na.omit(airquality)$Ozone, predict(ozone_nnet, type = "raw"))
abline(a = 0, b = 1)
```

`mlQda` *Supervised classification using quadratic discriminant analysis*

Description

Unified (formula-based) interface version of the quadratic discriminant analysis algorithm provided by `MASS::qda()`.

Usage

```
mlQda(train, ...)

ml_qda(train, ...)

## S3 method for class 'formula'
mlQda(formula, data, ..., subset, na.action)

## Default S3 method:
mlQda(train, response, ...)
```

```
## S3 method for class 'mlQda'
predict(
  object,
  newdata,
  type = c("class", "membership", "both"),
  prior = object$prior,
  method = c("plug-in", "predictive", "debiased", "looCV", "cv"),
  ...
)
```

Arguments

train	a matrix or data frame with predictors.
...	further arguments passed to MASS::qda() or its predict() method (see the corresponding help page).
formula	a formula with left term being the factor variable to predict and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the class ~ . short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using I()).
data	a data.frame to use as a training set.
subset	index vector with the cases to define the training set in use (this argument must be named, if provided).
na.action	function to specify the action to be taken if NAs are found. For ml_qda() na.fail is used by default. The calculation is stopped if there is any NA in the data. Another option is na.omit, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the predict() method, the default, and most suitable option, is na.exclude. In that case, rows with NAs in newdata= are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as newdata=).
response	a vector of factor for the classification.
object	an **mlQda** object
newdata	a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.
type	the type of prediction to return. "class" by default, the predicted classes. Other options are "membership" the membership (a number between 0 and 1) to the different classes, or "both" to return classes and memberships.
prior	the prior probabilities of class membership. By default, the prior are obtained from the object and, if they where not changed, correspond to the proportions observed in the training set.
method	"plug-in", "predictive", "debiased", "looCV", or "cv". "plug-in" (default) the usual unbiased parameter estimates are used. With "predictive",

the parameters are integrated out using a vague prior. With "debiased", an unbiased estimator of the log posterior probabilities is used. With "looCV", the leave-one-out cross-validation fits to the original data set are computed and returned. With "cv", cross-validation is used instead. If you specify method = "cv" then cvpredict() is used and you cannot provide newdata= in that case.

Value

ml_qda()/mlQda() creates an **mlQda**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also MASS::qda() that actually does the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_qda <- ml_qda(data = iris_train, Species ~ .)
summary(iris_qda)
confusion(iris_qda)
confusion(predict(iris_qda, newdata = iris_test), iris_test$Species)

# Another dataset (binary predictor... not optimal for qda, just for test)
data("HouseVotes84", package = "mlbench")
house_qda <- ml_qda(data = HouseVotes84, Class ~ ., na.action = na.omit)
summary(house_qda)
```

---

mlRforest *Supervised classification and regression using random forest*

Description

Unified (formula-based) interface version of the random forest algorithm provided by randomForest::randomForest().

Usage

```
mlRforest(train, ...)

ml_rforest(train, ...)

## S3 method for class 'formula'
mlRforest(
  formula,
  data,
  ntree = 500,
  mtry,
  replace = TRUE,
  classwt = NULL,
  ...,
  subset,
  na.action
)

## Default S3 method:
mlRforest(
  train,
  response,
  ntree = 500,
  mtry,
  replace = TRUE,
  classwt = NULL,
  ...
)

## S3 method for class 'mlRforest'
predict(
  object,
  newdata,
  type = c("class", "membership", "both", "vote"),
  method = c("direct", "oob", "cv"),
  ...
)
```

Arguments

`train`	a matrix or data frame with predictors.
`...`	further arguments passed to `randomForest::randomForest()` or its `predict()` method. There are many more arguments, see the corresponding help page.
`formula`	a formula with left term being the factor variable to predict (for supervised classification), a vector of numbers (for regression) or nothing (for unsupervised classification) and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the `class ~ .` short version

(that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using `I()`).

`data` a data.frame to use as a training set.

`ntree` the number of trees to generate (use a value large enough to get at least a few predictions for each input row). Default is 500 trees.

`mtry` number of variables randomly sampled as candidates at each split. Note that the default values are different for classification (sqrt(p) where p is number of variables in x) and regression (p/3)?

`replace` sample cases with or without replacement (`TRUE` by default)?

`classwt` priors of the classes. Need not add up to one. Ignored for regression.

`subset` index vector with the cases to define the training set in use (this argument must be named, if provided).

`na.action` function to specify the action to be taken if `NA`s are found. For `ml_rforest()` `na.fail` is used by default. The calculation is stopped if there is any `NA` in the data. Another option is `na.omit`, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the `predict()` method, the default, and most suitable option, is `na.exclude`. In that case, rows with `NA`s in `newdata=` are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as `newdata=`).

`response` a vector of factor (classification) or numeric (regression), or `NULL` (unsupervised classification).

`object` an **mlRforest** object

`newdata` a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.

`type` the type of prediction to return. `"class"` by default, the predicted classes. Other options are `"membership"` the membership (number between 0 and 1) to the different classes as assessed by the number of neighbors of these classes, or `"both"` to return classes and memberships. One can also use `"vote"`, which returns the number of trees that voted for each class.

`method` `"direct"` (default), `"oob"` or `"cv"`. `"direct"` predicts new cases in `newdata=` if this argument is provided, or the cases in the training set if not. Take care that not providing `newdata=` means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances (in the case of Random Forest, these metrics would most certainly falsely indicate a perfect classifier). Either use a different data set in `newdata=` or use the alternate approaches: out-of-bag (`"oob"`) or cross-validation ("cv"). The out-of-bag approach uses individuals that are not used to build the trees to assess performances. It is an unbiased estimates. If you specify `method = "cv"` then `cvpredict()` is used and you cannot provide `newdata=` in that case.

Value

ml_rforest()/mlRforest() creates an **mlRforest**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also randomForest::randomForest() that actually does the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_rf <- ml_rforest(data = iris_train, Species ~ .)
summary(iris_rf)
plot(iris_rf) # Useful to look at the effect of ntree=
# For such a relatively simple case, 50 trees are enough
iris_rf <- ml_rforest(data = iris_train, Species ~ ., ntree = 50)
summary(iris_rf)
predict(iris_rf) # Default type is class
predict(iris_rf, type = "membership")
predict(iris_rf, type = "both")
predict(iris_rf, type = "vote")
# Out-of-bag prediction (unbiased)
predict(iris_rf, method = "oob")
# Self-consistency (always very high for random forest, biased, do not use!)
confusion(iris_rf)
# This one is better
confusion(iris_rf, method = "oob") # Out-of-bag performances
# Cross-validation prediction is also a good choice when there is no test set
predict(iris_rf, method = "cv")  # Idem: cvpredict(res)
# Cross-validation for performances estimation
confusion(iris_rf, method = "cv")
# Evaluation of performances using a separate test set
confusion(predict(iris_rf, newdata = iris_test), iris_test$Species)

# Regression using random forest (from ?randomForest)
set.seed(131) # Useful for reproducibility (use a different number each time)
ozone_rf <- ml_rforest(data = airquality, Ozone ~ ., mtry = 3,
  importance = TRUE, na.action = na.omit)
summary(ozone_rf)
# Show "importance" of variables: higher value mean more important variables
round(randomForest::importance(ozone_rf), 2)
```

```
plot(na.omit(airquality)$Ozone, predict(ozone_rf))
abline(a = 0, b = 1)

# Unsupervised classification using random forest (from ?randomForest)
set.seed(17)
iris_urf <- ml_rforest(train = iris[, -5]) # Use only quantitative data
summary(iris_urf)
randomForest::MDSplot(iris_urf, iris$Species)
plot(stats::hclust(stats::as.dist(1 - iris_urf$proximity),
  method = "average"), labels = iris$Species)
```

`mlRpart` *Supervised classification and regression using recursive partitioning*

Description

Unified (formula-based) interface version of the recursive partitioning algorithm as implemented in `rpart::rpart()`.

Usage

```
mlRpart(train, ...)

ml_rpart(train, ...)

## S3 method for class 'formula'
mlRpart(formula, data, ..., subset, na.action)

## Default S3 method:
mlRpart(train, response, ..., .args. = NULL)

## S3 method for class 'mlRpart'
predict(
  object,
  newdata,
  type = c("class", "membership", "both"),
  method = c("direct", "cv"),
  ...
)
```

Arguments

`train`	a matrix or data frame with predictors.
`...`	further arguments passed to `rpart::rpart()` or its `predict()` method (see the corresponding help page.

formula — a formula with left term being the factor variable to predict (for supervised classification), a vector of numbers (for regression) and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the `class ~ .` short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using `I()`).

data — a data.frame to use as a training set.

subset — index vector with the cases to define the training set in use (this argument must be named, if provided).

na.action — function to specify the action to be taken if NAs are found. For `ml_rpart()` `na.fail` is used by default. The calculation is stopped if there is any NA in the data. Another option is `na.omit`, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the `predict()` method, the default, and most suitable option, is `na.exclude`. In that case, rows with NAs in `newdata=` are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as `newdata=`).

response — a vector of factor (classification) or numeric (regression).

.args. — used internally, do not provide anything here.

object — an **mlRpart** object

newdata — a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.

type — the type of prediction to return. `"class"` by default, the predicted classes. Other options are `"membership"` the membership (number between 0 and 1) to the different classes, or `"both"` to return classes and memberships,

method — `"direct"` (default) or `"cv"`. `"direct"` predicts new cases in `newdata=` if this argument is provided, or the cases in the training set if not. Take care that not providing `newdata=` means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different data set in `newdata=` or use the alternate cross-validation ("cv") technique. If you specify `method = "cv"` then `cvpredict()` is used and you cannot provide `newdata=` in that case.

Value

`ml_rpart()`/`mlRpart()` creates an **mlRpart**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like `predict()` or `cvpredict()`. In case you want to program new functions or extract specific components, inspect the "unclassed" object using `unclass()`.

See Also

`mlearning()`, `cvpredict()`, `confusion()`, also `rpart::rpart()` that actually does the classification.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_rpart <- ml_rpart(data = iris_train, Species ~ .)
summary(iris_rpart)
# Plot the decision tree for this classifier
plot(iris_rpart, margin = 0.03, uniform = TRUE)
text(iris_rpart, use.n = FALSE)
# Predictions
predict(iris_rpart) # Default type is class
predict(iris_rpart, type = "membership")
predict(iris_rpart, type = "both")
# Self-consistency, do not use for assessing classifier performances!
confusion(iris_rpart)
# Cross-validation prediction is a good choice when there is no test set
predict(iris_rpart, method = "cv")  # Idem: cvpredict(res)
confusion(iris_rpart, method = "cv")
# Evaluation of performances using a separate test set
confusion(predict(iris_rpart, newdata = iris_test), iris_test$Species)
```

`mlSvm` *Supervised classification and regression using support vector machine*

Description

Unified (formula-based) interface version of the support vector machine algorithm provided by `e1071::svm()`.

Usage

```
mlSvm(train, ...)

ml_svm(train, ...)

## S3 method for class 'formula'
mlSvm(
  formula,
  data,
  scale = TRUE,
  type = NULL,
  kernel = "radial",
```

```
  classwt = NULL,
  ...,
  subset,
  na.action
)

## Default S3 method:
mlSvm(
  train,
  response,
  scale = TRUE,
  type = NULL,
  kernel = "radial",
  classwt = NULL,
  ...
)

## S3 method for class 'mlSvm'
predict(
  object,
  newdata,
  type = c("class", "membership", "both"),
  method = c("direct", "cv"),
  na.action = na.exclude,
  ...
)
```

Arguments

train	a matrix or data frame with predictors.
...	further arguments passed to the classification or regression method. See e1071::svm().
formula	a formula with left term being the factor variable to predict (for supervised classification), a vector of numbers (for regression) or nothing (for unsupervised classification) and the right term with the list of independent, predictive variables, separated with a plus sign. If the data frame provided contains only the dependent and independent variables, one can use the class ~ . short version (that one is strongly encouraged). Variables with minus sign are eliminated. Calculations on variables are possible according to usual formula convention (possibly protected by using I()).
data	a data.frame to use as a training set.
scale	are the variables scaled (so that mean = 0 and standard deviation = 1)? TRUE by default. If a vector is provided, it is applied to variables with recycling.
type	For ml_svm()/mlSvm(), the type of classification or regression machine to use. The default value of NULL uses "C-classification" if response variable is factor and eps-regression if it is numeric. It can also be "nu-classification" or "nu-regression". The "C" and "nu" versions are basically the same but with a different parameterisation. The range of C is from zero to infinity, while the

range for nu is from zero to one. A fifth option is "one_classification" that is specific to novelty detection (find the items that are different from the rest). For predict(), the type of prediction to return. "class" by default, the predicted classes. Other options are "membership" the membership (number between 0 and 1) to the different classes, or "both" to return classes and memberships.

kernel — the kernel used by svm, see e1071::svm() for further explanations. Can be "radial", "linear", "polynomial" or "sigmoid".

classwt — priors of the classes. Need not add up to one.

subset — index vector with the cases to define the training set in use (this argument must be named, if provided).

na.action — function to specify the action to be taken if NAs are found. For ml_svm() na.fail is used by default. The calculation is stopped if there is any NA in the data. Another option is na.omit, where cases with missing values on any required variable are dropped (this argument must be named, if provided). For the predict() method, the default, and most suitable option, is na.exclude. In that case, rows with NAs in newdata= are excluded from prediction, but reinjected in the final results so that the number of items is still the same (and in the same order as newdata=).

response — a vector of factor (classification) or numeric (regression).

object — an **mlSvm** object

newdata — a new dataset with same conformation as the training set (same variables, except may by the class for classification or dependent variable for regression). Usually a test set, or a new dataset to be predicted.

method — "direct" (default) or "cv". "direct" predicts new cases in newdata= if this argument is provided, or the cases in the training set if not. Take care that not providing newdata= means that you just calculate the **self-consistency** of the classifier but cannot use the metrics derived from these results for the assessment of its performances. Either use a different data set in newdata= or use the alternate cross-validation ("cv") technique. If you specify method = "cv" then cvpredict() is used and you cannot provide newdata= in that case.

Value

ml_svm()/mlSvm() creates an **mlSvm**, **mlearning** object containing the classifier and a lot of additional metadata used by the functions and methods you can apply to it like predict() or cvpredict(). In case you want to program new functions or extract specific components, inspect the "unclassed" object using unclass().

See Also

mlearning(), cvpredict(), confusion(), also e1071::svm() that actually does the calculation.

Examples

```
# Prepare data: split into training set (2/3) and test set (1/3)
data("iris", package = "datasets")
train <- c(1:34, 51:83, 101:133)
```

```
iris_train <- iris[train, ]
iris_test <- iris[-train, ]
# One case with missing data in train set, and another case in test set
iris_train[1, 1] <- NA
iris_test[25, 2] <- NA

iris_svm <- ml_svm(data = iris_train, Species ~ .)
summary(iris_svm)
predict(iris_svm) # Default type is class
predict(iris_svm, type = "membership")
predict(iris_svm, type = "both")
# Self-consistency, do not use for assessing classifier performances!
confusion(iris_svm)
# Use an independent test set instead
confusion(predict(iris_svm, newdata = iris_test), iris_test$Species)

# Another dataset
data("HouseVotes84", package = "mlbench")
house_svm <- ml_svm(data = HouseVotes84, Class ~ ., na.action = na.omit)
summary(house_svm)
# Cross-validated confusion matrix
confusion(cvpredict(house_svm), na.omit(HouseVotes84)$Class)

# Regression using support vector machine
data(airquality, package = "datasets")
ozone_svm <- ml_svm(data = airquality, Ozone ~ ., na.action = na.omit)
summary(ozone_svm)
plot(na.omit(airquality)$Ozone, predict(ozone_svm))
abline(a = 0, b = 1)
```

`plot.confusion` *Plot a confusion matrix*

Description

Several graphical representations of **confusion** objects are possible: an image of the matrix with colored squares, a barplot comparing recall and precision, a stars plot also comparing two metrics, possibly also comparing two different classifiers of the same dataset, or a dendrogram grouping the classes relative to the errors observed in the confusion matrix (classes with more errors are pooled together more rapidly).

Usage

```
## S3 method for class 'confusion'
plot(
  x,
  y = NULL,
  type = c("image", "barplot", "stars", "dendrogram"),
  stat1 = "Recall",
```

```
  stat2 = "Precision",
  names,
  ...
)

confusion_image(
  x,
  y = NULL,
  labels = names(dimnames(x)),
  sort = "ward.D2",
  numbers = TRUE,
  digits = 0,
  mar = c(3.1, 10.1, 3.1, 3.1),
  cex = 1,
  asp = 1,
  colfun,
  ncols = 41,
  col0 = FALSE,
  grid.col = "gray",
  ...
)

confusionImage(
  x,
  y = NULL,
  labels = names(dimnames(x)),
  sort = "ward.D2",
  numbers = TRUE,
  digits = 0,
  mar = c(3.1, 10.1, 3.1, 3.1),
  cex = 1,
  asp = 1,
  colfun,
  ncols = 41,
  col0 = FALSE,
  grid.col = "gray",
  ...
)

confusion_barplot(
  x,
  y = NULL,
  col = c("PeachPuff2", "green3", "lemonChiffon2"),
  mar = c(1.1, 8.1, 4.1, 2.1),
  cex = 1,
  cex.axis = cex,
  cex.legend = cex,
  main = "F-score (precision versus recall)",
```

```
  numbers = TRUE,
  min.width = 17,
  ...
)

confusionBarplot(
  x,
  y = NULL,
  col = c("PeachPuff2", "green3", "lemonChiffon2"),
  mar = c(1.1, 8.1, 4.1, 2.1),
  cex = 1,
  cex.axis = cex,
  cex.legend = cex,
  main = "F-score (precision versus recall)",
  numbers = TRUE,
  min.width = 17,
  ...
)

confusion_stars(
  x,
  y = NULL,
  stat1 = "Recall",
  stat2 = "Precision",
  names,
  main,
  col = c("green2", "blue2", "green4", "blue4"),
  ...
)

confusionStars(
  x,
  y = NULL,
  stat1 = "Recall",
  stat2 = "Precision",
  names,
  main,
  col = c("green2", "blue2", "green4", "blue4"),
  ...
)

confusion_dendrogram(
  x,
  y = NULL,
  labels = rownames(x),
  sort = "ward.D2",
  main = "Groups clustering",
  ...
```

```
)

confusionDendrogram(
  x,
  y = NULL,
  labels = rownames(x),
  sort = "ward.D2",
  main = "Groups clustering",
  ...
)
```

Arguments

x	a **confusion** object
y	NULL (not used), or a second **confusion** object when two different classifications are compared in the plot ("stars" type).
type	the kind of plot to produce ("image", the default, or "barplot", "stars", "dendrogram").
stat1	the first metric to plot for the "stars" type (Recall by default).
stat2	the second metric to plot for the "stars" type (Precision by default).
names	names of the two classifiers to compare
...	further arguments passed to the function. It can be all arguments or the corresponding plot.
labels	labels to use for the two classifications. By default, they are the same as vars, or the one in the confusion matrix.
sort	are rows and columns of the confusion matrix sorted so that classes with larger confusion are closer together? Sorting is done using a hierarchical clustering with hclust(). The clustering method is "ward.D2" by default, but see the hclust() help for other options). If FALSE or NULL, no sorting is done.
numbers	are actual numbers indicated in the confusion matrix image?
digits	the number of digits after the decimal point to print in the confusion matrix. The default or zero leads to most compact presentation and is suitable for frequencies, but not for relative frequencies.
mar	graph margins.
cex	text magnification factor.
asp	graph aspect ratio. There is little reasons to change the default value of 1.
colfun	a function that calculates a series of colors, like e.g., cm.colors() that accepts one argument being the number of colors to be generated.
ncols	the number of colors to generate. It should preferably be 2 * number of levels + 1, where levels is the number of frequencies you want to evidence in the plot. Default to 41.
col0	should null values be colored or not (no, by default)?
grid.col	color to use for grid lines, or NULL for not drawing grid lines.

col	color(s) to use for the plot.
cex.axis	idem for axes. If NULL, the axis is not drawn.
cex.legend	idem for legend text. If NULL, no legend is added.
main	main title of the plot.
min.width	minimum bar width required to add numbers.

Value

Data calculate to create the plots are returned invisibly. These functions are mostly used for their side-effect of producing a plot.

Examples

```
data("Glass", package = "mlbench")
# Use a little bit more informative labels for Type
Glass$Type <- as.factor(paste("Glass", Glass$Type))

# Use learning vector quantization to classify the glass types
# (using default parameters)
summary(glass_lvq <- ml_lvq(Type ~ ., data = Glass))

# Calculate cross-validated confusion matrix and plot it in different ways
(glass_conf <- confusion(cvpredict(glass_lvq), Glass$Type))
# Raw confusion matrix: no sort and no margins
print(glass_conf, sums = FALSE, sort = FALSE)
# Plots
plot(glass_conf) # Image by default
plot(glass_conf, sort = FALSE) # No sorting
plot(glass_conf, type = "barplot")
plot(glass_conf, type = "stars")
plot(glass_conf, type = "dendrogram")

# Build another classifier and make a comparison
summary(glass_naive_bayes <- ml_naive_bayes(Type ~ ., data = Glass))
(glass_conf2 <- confusion(cvpredict(glass_naive_bayes), Glass$Type))

# Comparison plot for two classifiers
plot(glass_conf, glass_conf2)
```

---

prior *Get or set priors on a confusion matrix*

Description

Most metrics in supervised classifications are sensitive to the relative proportion of the items in the different classes. When a confusion matrix is calculated on a test set, it uses the proportions observed on that test set. If they are representative of the proportions in the population, metrics are not biased. When it is not the case, priors of a **confusion** object can be adjusted to better reflect proportions that are supposed to be observed in the different classes in order to get more accurate metrics.

Usage

```
prior(object, ...)

## S3 method for class 'confusion'
prior(object, ...)

prior(object, ...) <- value

## S3 replacement method for class 'confusion'
prior(object, ...) <- value
```

Arguments

object	a **confusion** object (or another class if a method is implemented)
...	further arguments passed to methods
value	a (named) vector of positive numbers of zeros of the same length as the number of classes in the **confusion** object. It can also be a single >= 0 number and in this case, equal probabilities are applied to all the classes (use 1 for relative frequencies and 100 for relative frequencies in percent). If the value has zero length or is NULL, original prior probabilities (from the test set) are used. If the vector is named, names must correspond to existing class names in the **confusion** object.

Value

prior() returns the current class frequencies associated with the first classification tabulated in the **confusion** object, i.e., for rows in the confusion matrix.

See Also

confusion()

Examples

```
data("Glass", package = "mlbench")
# Use a little bit more informative labels for Type
Glass$Type <- as.factor(paste("Glass", Glass$Type))
# Use learning vector quantization to classify the glass types
# (using default parameters)
summary(glass_lvq <- ml_lvq(Type ~ ., data = Glass))

# Calculate cross-validated confusion matrix
(glass_conf <- confusion(cvpredict(glass_lvq), Glass$Type))

# When the probabilities in each class do not match the proportions in the
# training set, all these calculations are useless. Having an idea of
# the real proportions (so-called, priors), one should first reweight the
# confusion matrix before calculating statistics, for instance:
prior1 <- c(10, 10, 10, 100, 100, 100) # Glass types 1-3 are rare
prior(glass_conf) <- prior1
glass_conf
```

```
summary(glass_conf, type = c("Fscore", "Recall", "Precision"))

# This is very different than if glass types 1-3 are abundants!
prior2 <- c(100, 100, 100, 10, 10, 10) # Glass types 1-3 are abundants
prior(glass_conf) <- prior2
glass_conf
summary(glass_conf, type = c("Fscore", "Recall", "Precision"))

# Weight can also be used to construct a matrix of relative frequencies
# In this case, all rows sum to one
prior(glass_conf) <- 1
print(glass_conf, digits = 2)
# However, it is easier to work with relative frequencies in percent
# and one gets a more compact presentation
prior(glass_conf) <- 100
glass_conf

# To reset row class frequencies to original propotions, just assign NULL
prior(glass_conf) <- NULL
glass_conf
prior(glass_conf)
```

`response` *Get the response variable for a mlearning object*

Description

The response is either the class to be predicted for a classification problem (and it is a factor), or the dependent variable in a regression model (and it is numeric in that case). For unsupervised classification, response is not provided and should return `NULL`.

Usage

```
response(object, ...)

## Default S3 method:
response(object, ...)
```

Arguments

`object`	an object having a response variable.
`...`	further parameter (depends on the method).

Value

The response variable of the training set, or `NULL` for unsupervised classification.

See Also

`mlearning()`, `train()`, `confusion()`

Examples

```
data("HouseVotes84", package = "mlbench")
house_rf <- ml_rforest(data = HouseVotes84, Class ~ .)
house_rf
response(house_rf)
```

`train` *Get the training variable for a mlearning object*

Description

The training variables (train) are the variables used to train a classifier, excepted the prediction (class or dependent variable).

Usage

```
train(object, ...)

## Default S3 method:
train(object, ...)
```

Arguments

`object`	an object having a train attribute.
`...`	further parameter (depends on the method).

Value

A data frame containing the training variables of the model.

See Also

`mlearning()`, `response()`, `confusion()`

Examples

```
data("HouseVotes84", package = "mlbench")
house_rf <- ml_rforest(data = HouseVotes84, Class ~ .)
house_rf
train(house_rf)
```

Index