Package 'MuMIn'

April 25, 2018

Type Package

Title Multi-Model Inference

Version 1.40.4

Date 2018-01-30

Encoding UTF-8

Author Kamil Bartoń

Maintainer Kamil Bartoń <kamil.barton@go2.pl>

Description Tools for performing model selection and model averaging. Automated model selection through subsetting the maximum model, with optional constraints for model inclusion. Model parameter and prediction averaging based on model weights derived from information criteria (AICc and alike) or custom model weighting schemes.

License GPL-2

Depends R (>= 3.0.0)

Imports graphics, methods, Matrix, stats, stats4

Suggests lme4 (>= 1.1.0), nlme, mgcv (>= 1.7.5), gamm4, MASS, nnet, survival (>= 2.37.5), geepack

Enhances aod, aods3, betareg, caper, coxme, cplm, gee, glmmML, logistf, MCMCglmm, ordinal, pscl, spdep, splm, unmarked (>= 0.12.2), geeM (>= 0.7.5)

LazyData yes

ByteCompile yes

R topics documented:

`MuMIn-package` *Multi-model inference*

Description

The package **MuMIn** contains functions to streamline information-theoretic model selection and carry out model averaging based on the information criteria.

Details

The collection of functions includes:

`dredge` performs automated model selection with subsets of the supplied 'global' model, and optional choices of other model properties (such as different link functions). The set of models may be generated either with 'all possible' combinations, or tailored according to the conditions specified.

`pdredge` does the same, but can parallelize model fitting process using a cluster.

`model.sel` creates a model selection table from hand-picked models.

`model.avg` calculates model averaged parameters, with standard errors and confidence intervals.

`AICc` calculates second-order Akaike information criterion. Some other criterions are provided, see below.

`stdize`, `stdizeFit`, `std.coef`, `partial.sd` can be used for standardization of data and model coefficients by Standard Deviation or Partial Standard Deviation.

For a complete list of functions, use `library(help = "MuMIn")`.

By default, AIC_c is used to rank the models and to obtain model weights, though any other information criteria can be utilised. At least the following ones are currently implemented in R: `AIC` and `BIC` in package **stats**, and `QAIC`, `QAICc`, `ICOMP`, `CAICF`, and Mallows' Cp in **MuMIn**. There is also `DIC` extractor for MCMC models, and `QIC` for GEE.

Most of R's common modelling functions are supported, for a full inventory see list of supported models.

Apart from the "regular" information criteria, model averaging can be performed using various types of model weighting algorithms: Bates-Granger, bootstrapped, cos-squared, jackknife, stacking, or ARM. These weighting functions apply mostly to `glm`s.

Author(s)

Kamil Bartoń

References

Burnham, K. P. and Anderson, D. R (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed. New York, Springer-Verlag.

See Also

`AIC`, `step` or `stepAIC` for stepwise model selection by AIC.

Examples

```
options(na.action = "na.fail") #  change the default "na.omit" to prevent models
                               #  from being fitted to different datasets in
                               #  case of missing values.

fm1 <- lm(y ~ ., data = Cement)
ms1 <- dredge(fm1)

# Visualize the model selection table:

par(mar = c(3,5,6,4))
plot(ms1, labAsExpr = TRUE)

model.avg(ms1, subset = delta < 4)

confset.95p <- get.models(ms1, cumsum(weight) <= .95)
avgmod.95p <- model.avg(confset.95p)
summary(avgmod.95p)
confint(avgmod.95p)
```

AICc *Second-order Akaike Information Criterion*

Description

Calculate Second-order Akaike Information Criterion for one or several fitted model objects (AIC_c, AIC for small samples).

Usage

```
AICc(object, ..., k = 2, REML = NULL)
```

Arguments

`object`	a fitted model object for which there exists a `logLik` method, or a `"logLik"` object.
`...`	optionally more fitted model objects.
`k`	the 'penalty' per parameter to be used; the default `k = 2` is the classical AIC.
`REML`	optional logical value, passed to the `logLik` method indicating whether the restricted log-likelihood or log-likelihood should be used. The default is to use the method used for model estimation.

Value

If just one object is provided, returns a numeric value with the corresponding AIC_c; if more than one object are provided, returns a `data.frame` with rows corresponding to the objects and columns representing the number of parameters in the model (*df*) and AIC_c.

Note

AIC_c should be used instead AIC when sample size is small in comparison to the number of estimated parameters (Burnham & Anderson 2002 recommend its use when $n/K < 40$).

Author(s)

Kamil Bartoń

References

Burnham, K. P. and Anderson, D. R (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed. New York, Springer-Verlag.

Hurvich, C. M. and Tsai, C.-L. (1989) Regression and time series model selection in small samples, *Biometrika* 76: 297–307.

See Also

Akaike's An Information Criterion: `AIC`

Other implementations: `AICc` in package **AICcmodavg**, `AICc` in package **bbmle** and `aicc` in package **glmulti**

Examples

```
#Model-averaging mixed models

options(na.action = "na.fail")

data(Orthodont, package = "nlme")

# Fit model by REML
fm2 <- lme(distance ~ Sex*age, data = Orthodont,
    random = ~ 1|Subject / Sex, method = "REML")

# Model selection: ranking by AICc using ML
ms2 <- dredge(fm2, trace = TRUE, rank = "AICc", REML = FALSE)

(attr(ms2, "rank.call"))

# Get the models (fitted by REML, as in the global model)
fmList <- get.models(ms2, 1:4)

# Because the models originate from 'dredge(..., rank = AICc, REML = FALSE)',
# the default weights in 'model.avg' are ML based:
summary(model.avg(fmList))

## Not run:
# the same result:
model.avg(fmList, rank = "AICc", rank.args = list(REML = FALSE))

## End(Not run)
```

`arm.glm` *Adaptive Regression by Mixing*

Description

Combine all-subsets GLMs using the ARM algorithm. Calculate ARM weights for a set of models.

Usage

```
arm.glm(object, R = 250, weight.by = c("aic", "loglik"), trace = FALSE)

armWeights(object, ..., data, weight.by = c("aic", "loglik"), R = 1000)
```

Arguments

`object`	for `arm.glm`, a fitted "global" `glm` object. For `armWeights`, a fitted `glm` object, or a `list` of such, or an `"averaging"` object.
`...`	more fitted model objects.
`R`	number of permutations.
`weight.by`	indicates whether model weights should be calculated with AIC or log-likelihood.
`trace`	if `TRUE`, information is printed during the running of `arm.glm`.
`data`	a data frame in which to look for variables for use with prediction. If omitted, the fitted linear predictors are used.

Details

For each of all-subsets of the "global" model, parameters are estimated using randomly sampled half of the data. Log-likelihood given the remaining half of the data is used to calculate AIC weights. This is repeated `R` times and mean of the weights is used to average all-subsets parameters estimated using complete data.

Value

`arm.glm` returns an object of class `"averaging"` contaning only "full" averaged coefficients. See `model.avg` for object description.

`armWeights` returns a numeric vector of model weights.

Note

Number of parameters is limited to `floor(nobs(object) / 2) - 1`. All-subsets respect marginality constraints.

Author(s)

Kamil Bartoń

References

Yang Y. (2001) Adaptive Regression by Mixing. *Journal of the American Statistical Association* 96: 574–588.

Yang Y. (2003) Regression with multiple candidate models: selecting or mixing? *Statistica Sinica* 13: 783–810.

See Also

`model.avg`, `par.avg`

`Weights` for assigning new model weights to an `"averaging"` object.

Other implementation of ARM algorithm: `arms` in (archived) package **MMIX**.

Other kinds of model weights: `BGWeights`, `bootWeights`, `cos2Weights`, `jackknifeWeights`, `stackingWeights`.

Examples

```
fm <- glm(y ~ X1 + X2 + X3 + X4, data = Cement)

summary(am1 <- arm.glm(fm, R = 15))

mst <- dredge(fm)

am2 <- model.avg(mst, fit = TRUE)

Weights(am2) <- armWeights(am2, data = Cement, R = 15)

# differences are due to small R:
coef(am1, full = TRUE)
coef(am2, full = TRUE)
```

`Beetle` *Flour beetle mortality data*

Description

Mortality of flour beetles (*Tribolium confusum*) due to exposure to gaseous carbon disulfide CS_2, from Bliss (1935).

Usage

```
Beetle
```

Format

`Beetle` is a data frame with 5 elements.

Prop a matrix with two columns named **nkilled** and **nsurvived**

mortality observed mortality rate

dose the dose of CS_2 in mg/L

n.tested number of beetles tested

n.killed number of beetles killed.

Source

Bliss C. I. (1935) The calculation of the dosage-mortality curve. *Annals of Applied Biology*, 22: 134-167.

References

Burnham, K. P. and Anderson, D. R. (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed. New York, Springer-Verlag.

Examples

```
# "Logistic regression example"
# from Burnham & Anderson (2002) chapter 4.11
# Fit a global model with all the considered variables

globmod <- glm(Prop ~ dose + I(dose^2) + log(dose) + I(log(dose)^2),
   data = Beetle, family = binomial, na.action = na.fail)
# A logical expression defining the subset of models to use:
# * either log(dose) or dose
# * the quadratic terms can appear only together with linear terms
msubset <- expression(xor(dose, `log(dose)`) &
    dc(dose, `I(dose^2)`) &
    dc(`log(dose)`, `I(log(dose)^2)`))

# Table 4.6
# Use 'varying' argument to fit models with different link functions
# Note the use of 'alist' rather than 'list' in order to keep the
# 'family' objects unevaluated
```

```
varying.link <- list(family = alist(
    logit = binomial("logit"),
    probit = binomial("probit"),
    cloglog = binomial("cloglog")
    ))

(ms12 <- dredge(globmod, subset = msubset, varying = varying.link,
    rank = AIC))

# Table 4.7 "models justifiable a priori"
(ms3 <- subset(ms12, has(dose, !`I(dose^2)`)))
# The same result, but would fit the models again:
# ms3 <- update(ms12, update(globmod, . ~ dose), subset =,
#    fixed = ~dose)
mod3 <- get.models(ms3, 1:3)
# Table 4.8. Predicted mortality probability at dose 40.
# calculate confidence intervals on logit scale
logit.ci <- function(p, se, quantile = 2) {
    C. <- exp(quantile * se / (p * (1 - p)))
    p /(p + (1 - p) * c(C., 1/C.))
}

mavg3 <- model.avg(mod3, revised.var = FALSE)
# get predictions both from component and averaged models
pred <- lapply(c(component = mod3, list(averaged = mavg3)), predict,
   newdata = list(dose = 40), type = "response", se.fit = TRUE)
# reshape predicted values
pred <- t(sapply(pred, function(x) unlist(x)[1:2]))
colnames(pred) <- c("fit", "se.fit")

# build the table
tab <- cbind(
    c(Weights(ms3), NA),
    pred,
    matrix(logit.ci(pred[,"fit"], pred[,"se.fit"],
        quantile = c(rep(1.96, 3), 2)), ncol = 2)
    )
colnames(tab) <- c("Akaike weight", "Predicted(40)", "SE", "Lower CI",
    "Upper CI")
rownames(tab) <- c(as.character(ms3$family), "model averaged")
print(tab, digits = 3, na.print = "")
# Figure 4.3
newdata <- list(dose = seq(min(Beetle$dose), max(Beetle$dose), length.out = 25))

# add model-averaged prediction with CI, using the same method as above
avpred <- predict(mavg3, newdata, se.fit = TRUE, type = "response")

avci <- matrix(logit.ci(avpred$fit, avpred$se.fit, quantile = 2), ncol = 2)


matplot(newdata$dose, sapply(mod3, predict, newdata, type = "response"),
    type = "l", xlab = quote(list("Dose of" ~ CS[2],(mg/L))),
    ylab = "Mortality", col = 2:4, lty = 3, lwd = 1
)
matplot(newdata$dose, cbind(avpred$fit, avci), type = "l", add = TRUE,
    lwd = 1, lty = c(1, 2, 2), col = 1)
```

```
legend("topleft", NULL, c(as.character(ms3$family), expression(`averaged`
    %+-% CI)), lty = c(3, 3, 3, 1), col = c(2:4, 1))
```

BGWeights *Bates-Granger model weights*

Description

Computes empirical weights based on out of sample forecast variances, following Bates and Granger (1969).

Usage

```
BGWeights(object, ..., data, force.update = FALSE)
```

Arguments

`object, ...`	two or more fitted `glm` objects, or a `list` of such, or an `"averaging"` object.
`data`	a data frame containing the variables in the model.
`force.update`	if `TRUE`, the much less efficient method of updating `glm` function will be used rather than directly *via* `glm.fit`. This only applies to `glm`s, in case of other model types `update` is always used.

Details

Bates-Granger model weights are calculated using prediction covariance. To get the estimate of prediction covariance, the models are fitted to randomly selected half of `data` and prediction is done on the remaining half. These predictions are then used to compute the variance-covariance between models, Σ. Model weights are then calculated as $w_{BG} = (1'\Sigma^{-1}1)^{-1}1\Sigma^{-1}$, where 1 a vector of 1-s.

Bates-Granger model weights may be outside of the $[0, 1]$ range, which may cause the averaged variances to be negative. Apparently this method works best when data is large.

Value

The function returns a numeric vector of model weights.

Note

For matrix inversion, `ginv` from package **MASS** is more stable near singularities than `solve`. It will be used as a fallback if `solve` fails and **MASS** is available.

Author(s)

Carsten Dormann, Kamil Bartoń

References

Bates, J. M. & Granger, C. W. J. (1969) The combination of forecasts. *Journal of the Operational Research Society*, 20: 451-468.

See Also

Weights, model.avg

Other model.weights: bootWeights, cos2Weights, jackknifeWeights, stackingWeights

Examples

```
fm <- glm(Prop ~ mortality + dose, family = binomial, Beetle, na.action = na.fail)
models <- lapply(dredge(fm, evaluate = FALSE), eval)
ma <- model.avg(models)

# this produces warnings because of negative variances:
set.seed(78)
Weights(ma) <- BGWeights(ma, data = Beetle)
coefTable(ma, full = TRUE)

# SE for prediction is not reliable if some or none of coefficient's SE
# are available
predict(ma, data = test.data, se.fit = TRUE)
coefTable(ma, full = TRUE)
```

---

bootWeights *Bootstrap model weights*

Description

Computes model weights using bootstrap.

Usage

```
bootWeights(object, ..., R, rank = c("AICc", "AIC", "BIC"))
```

Arguments

object, ...	two or more fitted glm objects, or a list of such, or an "averaging" object.
R	the number of replicates.
rank	a character string, specifying the information criterion to use for model ranking. Defaults to AICc.

Details

The models are fitted repeatedly to a resampled data set and ranked using AIC-type criterion. The model weights represent the proportion of replicates when a model has the lowest IC value.

Value

The function returns a numeric vector of model weights.

Author(s)

Kamil Bartoń, Carsten Dormann

See Also

Weights, model.avg

Other model.weights: BGWeights, cos2Weights, jackknifeWeights, stackingWeights

Examples

```
# To speed up the bootstrap, use 'x = TRUE' so that model matrix is included
#     in the returned object
fm <- glm(Prop ~ mortality + dose, family = binomial, data = Beetle,
   na.action = na.fail, x = TRUE)

fml <- lapply(dredge(fm, eval = FALSE), eval)
am <- model.avg(fml)

Weights(am) <- bootWeights(am, data = Beetle, R = 25)

summary(am)
```

Cement *Cement hardening data*

Description

Cement hardening data from Woods et al (1932).

Usage

```
Cement
```

Format

Cement is a data frame with 5 variables. *x1-x4* are four predictor variables expressed as a percentage of weight.

y calories of heat evolved per gram of cement after 180 days of hardening

X1 calcium aluminate

X2 tricalcium silicate

X3 tetracalcium alumino ferrite

X4 dicalcium silicate.

Source

Woods H., Steinour H.H., Starke H.R. (1932) Effect of composition of Portland cement on heat evolved during hardening. *Industrial & Engineering Chemistry* 24, 1207-1214.

References

Burnham, K. P. and Anderson, D. R (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed. New York, Springer-Verlag.

`cos2Weights` *Cos-squared model weights*

Description

Calculates cos-squared model weights, following the algorithm outlined in the appendix of Garthwaite & Mubwandarikwa (2010).

Usage

```
cos2Weights(object, ..., data, eps = 1e-06, maxit = 100,
  predict.args = list())
```

Arguments

`object, ...`	two or more fitted `glm` objects, or a `list` of such, or an `"averaging"` object. Currently only `lm` and `glm` objects are accepted.
`data`	a test data frame in which to look for variables for use with prediction. If omitted, the fitted linear predictors are used.
`eps`	tolerance for determining convergence.
`maxit`	maximum number of iterations.
`predict.args`	optionally, a `list` of additional arguments to be passed to `predict`.

Value

The function returns a numeric vector of model weights.

Author(s)

Carsten Dormann, adapted by Kamil Bartoń

References

Garthwaite, P. H. and Mubwandarikwa, E. (2010) Selection of weights for weighted model averaging. *Australian & New Zealand Journal of Statistics*, 52: 363–382.

Dormann, C. et al. (*in prep.*) Model averaging in ecology.

See Also

`Weights`, `model.avg`

Other model.weights: `BGWeights`, `bootWeights`, `jackknifeWeights`, `stackingWeights`

Examples

```
fm <- lm(y ~ X1 + X2 + X3 + X4, Cement, na.action = na.fail)
# most efficient way to produce a list of all-subsets models
models <- lapply(dredge(fm, evaluate = FALSE), eval)
ma <- model.avg(models)

test.data <- Cement
```

```
Weights(ma) <- cos2Weights(models, data = test.data)
predict(ma, data = test.data)
```

---

dredge *Automated model selection*

Description

Generate a set of models with combinations (subsets) of fixed effect terms in the global model, with optional rules for model inclusion.

Usage

```
dredge(global.model, beta = c("none", "sd", "partial.sd"), evaluate = TRUE,
    rank = "AICc", fixed = NULL, m.lim = NULL, m.min, m.max, subset,
    trace = FALSE, varying, extra, ct.args = NULL, ...)

## S3 method for class 'model.selection'
print(x, abbrev.names = TRUE, warnings = getOption("warn") != -1L, ...)
```

Arguments

`global.model`	a fitted 'global' model object. See 'Details' for a list of supported types.
`beta`	indicates whether and how the coefficients estimates should be standardized, and must be one of `"none"`, `"sd"` or `"partial.sd"`. You can specify just the initial letter. `"none"` corresponds to unstandardized coefficients, `"sd"` and `"partial.sd"` to coefficients standardized by SD and Partial SD, respectively. For backwards compatibility, logical value is also accepted, `TRUE` is equivalent to `"sd"` and `FALSE` to `"none"`. See `std.coef`.
`evaluate`	whether to evaluate and rank the models. If `FALSE`, a list of unevaluated `calls` is returned.
`rank`	optional custom rank function (returning an information criterion) to be used instead `AICc`, e.g. `AIC`, `QAIC` or `BIC`. See 'Details'.
`fixed`	optional, either a single sided formula or a character vector giving names of terms to be included in all models. See 'Subsetting'.
`m.lim, m.max, m.min`	optionally, the limits `c(lower, upper)` for number of terms in a single model (excluding the intercept). An `NA` means no limit. See 'Subsetting'. Specifying limits as `m.min` and `m.max` is allowed for backward compatibility.
`subset`	logical expression describing models to keep in the resulting set. See 'Subsetting'.
`trace`	if `TRUE` or `1`, all calls to the fitting function are printed before actual fitting takes place. If `trace > 1`, a progress bar is displayed.
`varying`	optionally, a named list describing the additional arguments to vary between the generated models. Item names correspond to the arguments, and each item provides a list of choices (i.e. `list(arg1 = list(choice1, choice2, ...), ...)`). Complex elements in the choice list (such as `family` objects) should be either named (uniquely) or quoted (unevaluated, e.g. using `alist`, see `quote`), otherwise the result may be visually unpleasant. See example in `Beetle`.

extra	optional additional statistics to include in the result, provided as functions, function names or a list of such (best if named or quoted). Similarly as in rank argument, each function must accept fitted model object as an argument and return (a value coercible to) a numeric vector. These can be e.g. additional information criterions or goodness-of-fit statistics. The character strings "R^2" and "adjR^2" are treated in a special way, and will add a likelihood-ratio based R^2 and modified-R^2 respectively to the result (this is more efficient than using r.squaredLR directly).
x	a model.selection object, returned by dredge.
abbrev.names	should printed term names be abbreviated? (useful with complex models).
warnings	if TRUE, errors and warnings issued during the model fitting are printed below the table (only with pdredge). To permanently remove the warnings, set the object's attribute "warnings" to NULL.
ct.args	optional list of arguments to be passed to coefTable (e.g. dispersion parameter for glm affecting standard errors used in subsequent model averaging).
...	optional arguments for the rank function. Any can be an unevaluatec expression, in which case any x within it will be substituted with a current model.

Details

Models are fitted through repeated evaluation of modified call extracted from the global.model (in a similar fashion as with update). This approach, while robust in that it can be applied to most model types, is inefficient because of its considerable computational overhead.

Note that the number of combinations grows exponentially with number of predictors (2^N, less when interactions are present, see below).

The fitted model objects are not stored in the result. To get (possibly a subset of) the models, use get.models on the object returned by dredge. Another way of getting all the models is to run lapply(dredge(..., evaluate = FALSE), eval), which avoids fitting the models twice.

For a list of model types that can be used as a global.model see list of supported models. Modelling functions not storing call in their result should be evaluated *via* the wrapper function created by updateable.

Information criterion: rank is found by a call to match.fun and may be specified as a function or a symbol or a character string specifying a function to be searched for from the environment of the call to dredge. The function rank must accept model object as its first argument and always return a scalar.

Interactions: By default, marginality constraints are respected, so "all possible combinations" include only those containing interactions with their respective main effects and all lower order terms. However, if global.model makes an exception to this principle (e.g. due to a nested design such as a / (b + d)), this will be reflected in the subset models.

Subsetting: There are three ways to constrain the resulting set of models: setting limits to the number of terms in a model with m.lim, binding term(s) to all models with fixed, and more complex rules can be applied using argument subset. To be included in the selection table, the model formulation must satisfy all these conditions.

subset can take either a form of an *expression* or a *matrix*. The latter should be a lower triangular matrix with logical values, where columns and rows correspond to global.model terms. Value subset["a", "b"] == FALSE will exclude any model containing both terms *a* and *b*. demo(dredge.subset) has examples of using the subset matrix in conjunction with correlation matrices to exclude models containing collinear predictors.

In the form of `expression`, the argument `subset` acts in a similar fashion to that in the function `subset` for `data.frames`: model terms can be referred to by name as variables in the expression, with the difference being that are interpreted as logical values (i.e. equal to `TRUE` if the term exists in the model).

There is also `.(x)` and `.(+x)` notation indicating, respectively, any and all interactions including a *term* `x`. It is only useful with marginality exceptions.

The expression can contain any of the `global.model` terms (`getAllTerms(global.model)` lists them), as well as names of the `varying` argument items. Names of `global.model` terms take precedence when identical to names of `varying`, so to avoid ambiguity `varying` variables in `subset` expression should be enclosed in `V()` (e.g. `subset = V(family) == "Gamma"` assuming that `varying` is something like `list(family = c(..., "Gamma"))`).

If item names in `varying` are missing, the items themselves are coerced to names. Call and symbol elements are represented as character values (via `deparse`), and everything except numeric, logical, character and `NULL` values is replaced by item numbers (e.g. `varying = list(family = list(..., Gamma)` should be referred to as `subset = V(family) == 2`. This can quickly become confusing, therefore it is recommended to use named lists. `demo(dredge.varying)` provides examples.

The `subset` expression can also contain variable `` `*nvar*` `` (backtick-quoted), equal to number of terms in the model (**not** the number of estimated parameters).

To make inclusion of a model term conditional on presence of another model term, the function `dc` ("**d**ependency **c**hain") can be used in the `subset` expression. `dc` takes any number of term names as arguments, and allows a term to be included only if all preceding ones are also present (e.g. `subset = dc(a, b, c)` allows for models `a`, `a+b` and `a+b+c` but not `b`, `c`, `b+c` or `a+c`).

`subset` expression can have a form of an unevaluated `call`, `expression` object, or a one sided `formula`. See 'Examples'.

Compound model terms (such as interactions, 'as-is' expressions within `I()` or smooths in `gam`) should be enclosed within curly brackets (e.g. `{s(x,k=2)}`), or backticks (like non-syntactic names, e.g. `` `s(x, k = 2)` ``). Backticks-quoted names must match exactly (including whitespace) the term names as given by `getAllTerms`.

`subset` *expression syntax summary:*

`a & b` indicates that model terms *a* and *b* must be present (see Logical Operators)

`{log(x,2)}` **or** `` `log(x, 2)` `` represent a complex model term `log(x, 2)`

`V(x)` represents a `varying` variable *x*

`.(x)` indicates that at least one term containing the term *x* must be present

`.(+x)` indicates that all the terms containing the term *x* must be present

`dc(a, b, c,...)` 'dependency chain': *b* is allowed only if *a* is present, and *c* only if both *a* and *b* are present, etc.

`` `*nvar*` `` number of terms.

To simply keep certain terms in all models, use of argument `fixed` is much more efficient. The `fixed` formula is interpreted in the same manner as model formula and so the terms need not to be quoted.

Missing values: Use of `na.action = "na.omit"` (R's default) or `"na.exclude"` in `global.model` must be avoided, as it results with sub-models fitted to different data sets, if there are missing values. Error is thrown if it is detected.

It is a common mistake to give `na.action` as an argument in the call to `dredge` (typically resulting in an error from the `rank` function to which the argument is passed through '...'), while the correct way is either to pass `na.action` in the call to the global model or to set it as a global option.

Methods: There are `subset` and `plot` methods, the latter creates a graphical representation of model weights and variable relative importance. Coefficients can be extracted with `coef` or `coefTable`.

Value

An object of class `c("model.selection", "data.frame")`, being a `data.frame`, where each row represents one model. See `model.selection.object` for its structure.

Note

Users should keep in mind the hazards that a "thoughtless approach" of evaluating all possible models poses. Although this procedure is in certain cases useful and justified, it may result in selecting a spurious "best" model, due to the model selection bias.

"Let the computer find out" is a poor strategy and usually reflects the fact that the researcher did not bother to think clearly about the problem of interest and its scientific setting (Burnham and Anderson, 2002).

Author(s)

Kamil Bartoń

See Also

`pdredge` is a parallelized version of this function (uses a cluster).

`get.models`, `model.avg`. `model.sel` for manual model selection tables.

Possible alternatives: `glmulti` in package **glmulti** and `bestglm` (**bestglm**). `regsubsets` in package **leaps** also performs all-subsets regression.

Lasso variable selection provided by various packages, e.g. **glmnet**, **lars** or **glmmLasso**.

Examples

```
# Example from Burnham and Anderson (2002), page 100:

#  prevent fitting sub-models to different datasets

options(na.action = "na.fail")

fm1 <- lm(y ~ ., data = Cement)
dd <- dredge(fm1)
subset(dd, delta < 4)

# Visualize the model selection table:

par(mar = c(3,5,6,4))
plot(dd, labAsExpr = TRUE)


# Model average models with delta AICc < 4
model.avg(dd, subset = delta < 4)

#or as a 95% confidence set:
model.avg(dd, subset = cumsum(weight) <= .95) # get averaged coefficients

#'Best' model
summary(get.models(dd, 1)[[1]])

## Not run:
# Examples of using 'subset':
```

```
# keep only models containing X3
dredge(fm1, subset = ~ X3) # subset as a formula
dredge(fm1, subset = expression(X3)) # subset as expression object
# the same, but more effective:
dredge(fm1, fixed = "X3")
# exclude models containing both X1 and X2 at the same time
dredge(fm1, subset = !(X1 && X2))
# Fit only models containing either X3 or X4 (but not both);
# include X3 only if X2 is present, and X2 only if X1 is present.
dredge(fm1, subset = dc(X1, X2, X3) && xor(X3, X4))
# the same as above, without "dc"
dredge(fm1, subset = (X1 | !X2) && (X2 | !X3) && xor(X3, X4))

# Include only models with up to 2 terms (and intercept)
dredge(fm1, m.lim = c(0, 2))

## End(Not run)

# Add R^2 and F-statistics, use the 'extra' argument
dredge(fm1, m.lim = c(NA, 1), extra = c("R^2", F = function(x)
    summary(x)$fstatistic[[1]]))

# with summary statistics:
dredge(fm1, m.lim = c(NA, 1), extra = list(
    "R^2", "*" = function(x) {
        s <- summary(x)
        c(Rsq = s$r.squared, adjRsq = s$adj.r.squared,
            F = s$fstatistic[[1]])
    })
)

# Add other information criterions (but rank with AICc):
dredge(fm1, m.lim = c(NA, 1), extra = alist(AIC, BIC, ICOMP, Cp))
```

`exprApply` *Apply a function to calls inside an expression*

Description

Apply function `FUN` to each occurence of a call to `what()` (or a symbol `what`) in an unevaluated expression. It can be used for advanced manipulation of expressions. Intended primarily for internal use.

Usage

```
exprApply(expr, what, FUN, ..., symbols = FALSE)
```

Arguments

`expr`	an unevaluated expression.
`what`	character string giving the name of a function. Each call to `what` inside `expr` will be passed to `FUN`. `what` can be also a character representation of an operator or parenthesis (including curly and square brackets) as these are primitive functions in R. Set `what` to `NA` to match all names.

FUN	a function to be applied.
symbols	logical value controlling whether FUN should be applied to symbols as well as calls.
...	optional arguments to FUN.

Details

FUN is found by a call to match.fun and can be either a function or a symbol (e.g., a backquoted name) or a character string specifying a function to be searched for from the environment of the call to exprApply.

Value

A (modified) expression.

Note

If expr has a source reference information ("srcref" attribute), modifications done by exprApply will not be visible when printed unless srcref is removed. However, exprApply does remove source reference from any function expression inside expr.

Author(s)

Kamil Bartoń

See Also

Expression-related functions: substitute, expression, quote and bquote.

Similar function walkCode exists in package **codetools**.

Functions useful inside FUN: as.name, as.call, call, match.call etc.

Examples

```
### simple usage:
# print all Y(...) terms in a formula (note that symbol "Y" is omitted):
exprApply(~ X(1) + Y(2 + Y(4)) + N(Y + Y(3)), "Y", print)

# replace X() with log(X, base = n)
exprApply(expression(A() + B() + C()), c("A", "B", "C"), function(expr, base) {
    expr[[2]] <- expr[[1]]
    expr[[1]] <- as.name("log")
    expr$base <- base
    expr
}, base = 10)

###
# TASK: fit lm with two poly terms, varying the degree from 1 to 3 in each.
# lm(y ~ poly(X1, degree = a) + poly(X2, degree = b), data = Cement)
# for a = {1,2,3} and b = {1,2,3}

# First we create a wrapper function for lm. Within it, use "exprApply" to add
# "degree" argument to all occurences of "poly()" having "X1" or "X2" as the
# first argument. Values for "degree" are taken from arguments "d1" and "d2"
```

```
lmpolywrap <- function(formula, d1 = NA, d2 = NA, ...) {
    cl <- origCall <- match.call()
    cl[[1]] <- as.name("lm")
    cl$formula <- exprApply(formula, "poly", function(e, degree, x) {
        i <- which(e[[2]] == x)[1]
        if(!is.na(i) && !is.na(degree[i])) e$degree <- degree[i]
        e
    }, degree = c(d1, d2), x = c("X1", "X2"))
    cl$d1 <- cl$d2 <- NULL
    fit <- eval(cl, parent.frame())
    fit$call <- origCall # replace the stored call
    fit
}

# global model:
fm <- lmpolywrap(y ~ poly(X1) + poly(X2), data = Cement)

# Use "dredge" with argument "varying" to generate calls of all combinations of
# degrees for poly(X1) and poly(X2). Use "fixed = TRUE" to keep all global model
# terms in all models.
# Since "dredge" expects that global model has all the coefficients the
# submodels can have, which is not the case here, we first generate model calls,
# evaluate them and feed to "model.sel"

modCalls <- dredge(fm,
    varying = list(d1 = 1:3, d2 = 1:3),
    fixed = TRUE,
    evaluate = FALSE
)

model.sel(models <- lapply(modCalls, eval))

# Note: to fit *all* submodels replace "fixed = TRUE" with:
# "subset = (d1==1 || {poly(X1)}) && (d2==1 || {poly(X2)})"
# This is to avoid fitting 3 identical models when the matching "poly()" term is
# absent.
```

`Formula manipulation` *Manipulate model formulas*

Description

`simplify.formula` rewrites a `formula` using shorthand notation. Currently only the factor crossing operator `*` is applied, so that expanded expression such as `a+b+a:b` becomes `a*b`. `expand.formula` does the opposite, additionally expanding other expressions, i.e. all nesting (`/`), grouping and `^`.

Usage

```
simplify.formula(x)
expand.formula(x)
```

Arguments

`x`	a `formula` or an object from which it can be extracted (such as a fitted model object).

Author(s)

Kamil Bartoń

See Also

formula

delete.response, drop.terms, and reformulate

Examples

```
simplify.formula(y ~ a + b + a:b + (c + b)^2)
simplify.formula(y ~ a + b + a:b + 0)

expand.formula(~ a * b)
```

---

get.models *Retrieve models from selection table*

Description

Generate or extract a list of fitted model objects from a "model.selection" table, optionally using parallel computation in a cluster.

Usage

```
get.models(object, subset, cluster = NA, ...)
```

Arguments

object	object returned by dredge.
subset	subset of models, an expression evaluated within the model selection table (see 'Details').
cluster	optionally, a "cluster" object. If it is a valid cluster, models are evaluated using parallel computation.
...	additional arguments to update the models. For example, in lme one may want to use method = "REML" while using "ML" for model selection.

Details

The argument subset must be explicitely provided. This is to assure that a potentially long list of models is not fitted unintentionally. To evaluate all models, set subset to NA or TRUE.

If subset is a character vector, it is interpreted as names of rows to be selected.

Value

list of fitted model objects.

Note

Alternatively, `getCall` and `eval` can be used to compute a model out of the `"model.selection"` table (e.g. `eval(getCall(<model.selection>, i))`, where `i` is the model index or name).

Using `get.models` following `dredge` is not efficient as the requested models have to be fitted again. If the number of generated models is reasonable, consider using `lapply(dredge(..., evaluate = FALSE), eval)`, which generates a list of all model calls and evaluates them into a list of model objects. This avoids fitting the models twice.

`pget.models` is still available, but is deprecated.

Author(s)

Kamil Bartoń

See Also

`dredge` and `pdredge`, `model.avg`

`makeCluster` in packages **parallel** and **snow**

Examples

```
# Mixed models:


fm2 <- lme(distance ~ age + Sex, data = Orthodont,
    random = ~ 1 | Subject, method = "ML")
ms2 <- dredge(fm2)

# Get top-most models, but fitted by REML:
(confset.d4 <- get.models(ms2, subset = delta < 4, method = "REML"))

## Not run:
# Get the top model:
get.models(ms2, subset = 1)[[1]]

## End(Not run)
```

GPA *Grade Point Average data*

Description

First-year college Grade Point Average (GPA) from Graybill and Iyer (1994).

Usage

```
GPA
```

Format

GPA is a data frame with 5 variables. *y* is the first-year college Grade Point Average (GPA) and *x1-x4* are four predictor variables from standardized tests (SAT) administered before matriculation.

y GPA

x1 math score on the SAT

x2 verbal score on the SAT

x3 high school math

x4 high school English

Source

Graybill, F.A. and Iyer, H.K. (1994). *Regression analysis: concepts and applications*. Duxbury Press, Belmont, CA.

References

Burnham, K. P. and Anderson, D. R (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed. New York, Springer-Verlag.

---

importance *Relative variable importance*

Description

Sum of 'Akaike weights' over all models including the explanatory variable.

Usage

```
importance(x)
```

Arguments

x	either a list of fitted model objects, or a "model.selection" or "averaging" object.

Value

a numeric vector of so called relative importance values, named as the predictor variables.

Author(s)

Kamil Bartoń

See Also

Weights

dredge, model.avg, model.sel

Examples

```
# Generate some models
fm1 <- lm(y ~ ., data = Cement, na.action = na.fail)
ms1 <- dredge(fm1)

# Importance can be calculated/extracted from various objects:
importance(ms1)
## Not run:
importance(subset(model.sel(ms1), delta <= 4))
importance(model.avg(ms1, subset = delta <= 4))
importance(subset(ms1, delta <= 4))
importance(get.models(ms1, delta <= 4))

## End(Not run)

# Re-evaluate the importances according to BIC
# note that re-ranking involves fitting the models again

# 'nobs' is not used here for backwards compatibility
lognobs <- log(length(resid(fm1)))

importance(subset(model.sel(ms1, rank = AIC, rank.args = list(k = lognobs)),
    cumsum(weight) <= .95))

# This gives a different result than previous command, because 'subset' is
# applied to the original selection table that is ranked with 'AICc'
importance(model.avg(ms1, rank = AIC, rank.args = list(k = lognobs),
    subset = cumsum(weight) <= .95))
```

`Information criteria` *Various information criteria*

Description

Calculate Mallows' *Cp* and Bozdogan's ICOMP and CAIFC information criteria.

Extract or calculate Deviance Information Criterion from `MCMCglmm` and `merMod` object.

Usage

```
Cp(object, ..., dispersion = NULL)
ICOMP(object, ..., REML = NULL)
CAICF(object, ..., REML = NULL)
DIC(object, ...)
```

Arguments

`object`	a fitted model object (in case of ICOMP and CAICF, `logLik` and `vcov` methods must exist for the object). For `DIC`, an object of class `"MCMCglmm"` or `"merMod"`.
`...`	optionally more fitted model objects.
`dispersion`	the dispersion parameter. If `NULL`, it is inferred from object.

REML optional logical value, passed to the logLik method indicating whether the restricted log-likelihood or log-likelihood should be used. The default is to use the method used for model estimation.

Details

Mallows' *Cp* statistic is the residual deviance plus twice the estimate of σ^2 times the residual degrees of freedom. It is closely related to AIC (and a multiple of it if the dispersion is known).

ICOMP (I for informational and COMP for complexity) penalizes the covariance complexity of the model, rather than the number of parameters directly.

CAICF (C is for 'consistent' and F denotes the use of the Fisher information matrix) includes with penalty the natural logarithm of the determinant of the estimated Fisher information matrix.

Value

If just one object is provided, the functions return a numeric value with the corresponding IC; otherwise a data.frame with rows corresponding to the objects is returned.

References

Mallows, C. L. (1973) Some comments on *Cp*. *Technometrics* 15: 661–675.

Bozdogan, H. and Haughton, D.M.A. (1998) Information complexity criteria for regression models. *Comp. Stat. & Data Analysis* 28: 51-76.

Anderson, D. R. and Burnham, K. P. (1999) Understanding information criteria for selection among capture-recapture or ring recovery models. *Bird Study* 46: 14–21.

Spiegelhalter, D.J., Best, N.G., Carlin, B.R., van der Linde, A. (2002) Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society Series B-Statistical Methodology* 64: 583–616.

See Also

AIC and BIC in **stats**, AICc. QIC for GEE model selection. extractDIC in package **arm**, on which the (non-visible) method extractDIC.merMod used by DIC is based.

---

jackknifeWeights *Jackknifed model weights*

Description

Computes model weights optimized for jackknifed model fits.

Usage

```
jackknifeWeights(object, ..., data, type = c("loglik", "rmse"),
  family = NULL, weights = NULL, optim.method = "BFGS", maxit = 1000,
  optim.args = list(), start = NULL, force.update = FALSE,
  py.matrix = FALSE)
```

Arguments

object, ...	two or more fitted glm objects, or a list of such, or an "averaging" object.
data	a data frame containing the variables in the model. It is optional if all models are glm.
type	a character string specifying the function to minimize. Either "rmse" or "loglik".
family	used only if type = "loglik", a family object to be used for likelihood calculation. Not needed if all models share the same family and link function.
weights	an optional vector of 'prior weights' to be used in the model fitting process. Should be NULL or a numeric vector.
optim.method	optional, optimisation method, passed to optim.
maxit	optional, the maximum number of iterations, passed to optim.
optim.args	optional list of other arguments passed to optim.
start	starting values for model weights. Numeric of length equal the number of models.
force.update	for glm, the glm.fit function is used for fitting models to the train data, which is much more efficient. Set to TRUE to use update instead.
py.matrix	either a boolean value, then if TRUE a jackknifed prediction matrix is returned and if FALSE a vector of jackknifed model weights, or a $N \times M$ matrix (*number of cases* × *number of models*) that is interpreted as a jackknifed prediction matrix and it is used for optimisation (i.e. the jackknife procedure is skipped).

Details

Model weights are chosen (using optim) to minimise RMSE or log-likelihood of the prediction for data point i, of a model fitted omitting that data point i. The jackknife procedure is therefore run for all provided models and for all data points.

Value

The function returns a numeric vector of model weights.

Note

This procedure can give variable results depending on the optimisation method and starting values. It is therefore advisable to make several replicates using different optim.methods. See optim for possible values for this argument.

Author(s)

Kamil Bartoń. Carsten Dormann

References

Hansen, B. E. & Racine, J. S. (2012) Jackknife model averaging. *Journal of Econometrics*, 979: 38–46

See Also

Weights, model.avg

Other model.weights: BGWeights, bootWeights, cos2Weights, stackingWeights

Examples

```
fm <- glm(Prop ~ mortality * dose, binomial(), Beetle, na.action = na.fail)

fits <- lapply(dredge(fm, eval = FALSE), eval)

amJk <- amAICc <- model.avg(fits)
set.seed(666)
Weights(amJk) <- jackknifeWeights(fits, data = Beetle)

coef(amJk)
coef(amAICc)
```

---

loo *Leave-one-out cross-validation*

Description

Computes the RMSE/log-likelihood based on leave-one-out cross-validation.

Usage

```
loo(object, type = c("loglik", "rmse"), ...)
```

Arguments

`object`	a fitted object model, currently only `lm`/`glm` is accepted.
`type`	the criterion to use, given as a character string, either `"rmse"` for Root-Mean-Square Error or `"loglik"` for log-likelihood.
`...`	other arguments are currently ignored.

Details

Leave-one-out cross validation is a K-fold cross validation, with K equal to the number of data points in the set N. For all i from 1 to N, the model is fitted to all the data except for i-th row and a prediction is made for that value. The average error is computed and used to evaluate the model.

Value

`loo` returns a single numeric value of RMSE or mean log-likelihood.

Author(s)

Kamil Bartoń, based on code by Carsten Dormann

References

Dormann, C. et al. (*in prep.*) Model averaging in ecology.

Examples

```
fm <- lm(y ~ X1 + X2 + X3 + X4, Cement)
loo(fm, type = "l")
loo(fm, type = "r")

## Compare LOO_RMSE and AIC/c
options(na.action = na.fail)
dd <- dredge(fm, rank = loo, extra = list(AIC, AICc), type = "rmse")
plot(loo ~ AIC, dd, ylab = expression(LOO[RMSE]), xlab = "AIC/c")
points(loo ~ AICc, data = dd, pch = 19)
legend("topleft", legend = c("AIC", "AICc"), pch = c(1, 19))
```

---

merge.model.selection *Combine model selection tables*

Description

Combine two or more model selection tables.

Usage

```
## S3 method for class 'model.selection'
merge(x, y, suffixes = c(".x", ".y"), ...)

## S3 method for class 'model.selection'
rbind(..., deparse.level = 1, make.row.names = TRUE)
```

Arguments

x, y, ...	model.selection objects to be combined. (... ignored in merge)
suffixes	a character vector with two elements that are appended respectively to row names of the combined tables.
make.row.names	logical indicating if unique and valid row.names should be constructed from the arguments.
deparse.level	ignored.

Value

A "model.selection" object containing models from all provided tables.

Note

Both Δ_{IC} values and *Akaike weights* are recalculated in the resulting tables.

Models in the combined model selection tables must be comparable, i.e. fitted to the same data, however only very basic checking is done to verify that. The models must also be ranked by the same information criterion.

Unlike the merge method for data.frame, this method appends second table to the first (similarly to rbind).

Author(s)

Kamil Bartoń

See Also

dredge, model.sel, merge, rbind.

Examples

```
## Not run:
require(mgcv)

ms1 <- dredge(glm(Prop ~ dose + I(dose^2) + log(dose) + I(log(dose)^2),
    data = Beetle, family = binomial, na.action = na.fail))

fm2 <- gam(Prop ~ s(dose, k = 3), data = Beetle, family = binomial)

merge(ms1, model.sel(fm2))

## End(Not run)
```

Model utilities *Model utility functions*

Description

These functions extract or calculate various values from provided fitted model objects(s). They are mainly meant for internal use.

coeffs extracts model coefficients;

getAllTerms extracts independent variable names from a model object;

coefTable extracts a table of coefficients, standard errors and associated degrees of freedom when possible;

get.response extracts response variable from fitted model object;

model.names generates shorthand (alpha)numeric names for one or several fitted models.

Usage

```
coeffs(model)

getAllTerms(x, ...)
## S3 method for class 'terms'
getAllTerms(x, intercept = FALSE, offset = TRUE, ...)

coefTable(model, ...)
## S3 method for class 'averaging'
coefTable(model, full = FALSE, adjust.se = TRUE, ...)
## S3 method for class 'lme'
coefTable(model, adjustSigma, ...)
## S3 method for class 'gee'
coefTable(model, ..., type = c("naive", "robust"))
```

```
get.response(x, data = NULL, ...)

model.names(object, ..., labels = NULL, use.letters = FALSE)
```

Arguments

`model`	a fitted model object.
`object`	a fitted model object or a list of such objects.
`x`	a fitted model object or a `formula`.
`offset`	should 'offset' terms be included?
`intercept`	should terms names include the intercept?
`full, adjust.se`	logical, apply to `"averaging"` objects. If `full` is `TRUE`, the full model averaged coefficients are returned, and subset-averaged ones otherwise. If `adjust.se` is `TRUE`, inflated standard errors are returned. See 'Details' in `par.avg`.
`adjustSigma`	See `summary.lme`.
`type`	for GEE models, the type of covariance estimator to calculate returned standard errors on. Either `"naive"` or `"robust"` ('sandwich').
`labels`	optionally, a character vector with names of all the terms, e.g. from a global model. `model.names` enumerates the model terms in order of their appearance in the list and in the models. Therefore changing the order of the models leads to different names. Providing `labels` prevents that.
`...`	in `model.names`, more fitted model objects. In `coefTable` arguments that are passed to appropriate `vcov` or `summary` method (e.g. `dispersion` parameter for `glm` may be used here). In `get.response`, if `data` is given, arguments to be passed to `model.frame`. In other functions may be silently ignored.
`data`	a `data.frame`, `list` or `environment` (or object coercible to a `data.frame`), containing the variables in `x`. Required only if `x` is a `formula`, otherwise it can be used to get the response variable for a different data set.
`use.letters`	logical, whether letters should be used instead of numeric codes.

Details

The functions `coeffs`, `getAllTerms` and `coefTable` provide interface between the model object and `model.avg` (and `dredge`). Custom methods can be written to provide support for additional classes of models.

Note

`coeffs`'s value is in most cases identical to that returned by `coef`, the only difference being it returns fixed effects' coefficients for mixed models, and the value is always a named numeric vector.

Use of `tTable` is deprecated in favour of `coefTable`.

Author(s)

Kamil Bartoń

model.avg *Model averaging*

Description

Model averaging based on an information criterion.

Usage

```
model.avg(object, ..., revised.var = TRUE)

## Default S3 method:
model.avg(object, ..., beta = c("none", "sd", "partial.sd"),
    rank = NULL, rank.args = NULL, revised.var = TRUE,
    dispersion = NULL, ct.args = NULL)

## S3 method for class 'model.selection'
model.avg(object, subset, fit = FALSE, ..., revised.var = TRUE)
```

Arguments

object	a fitted model object or a list of such objects, or a "model.selection" object. See 'Details'.
...	for default method, more fitted model objects. Otherwise, arguments that are passed to the default method.
beta	indicates whether and how the component models' coefficients should be standardized. See the argument's description in dredge.
rank	optionally, a rank function (returning an information criterion) to use instead of AICc, e.g. BIC or QAIC, may be omitted if object is a model list returned by get.models or a "model.selection" object. See 'Details'.
rank.args	optional list of arguments for the rank function. If one is an expression, an x within it is substituted with a current model.
revised.var	logical, indicating whether to use revised formula for standard errors. See par.avg.
dispersion	the dispersion parameter for the family used. See summary.glm. This is used currently only with glm, is silently ignored otherwise.
ct.args	optional list of arguments to be passed to coefTable (besides dispersion).
subset	see subset method for "model.selection" object.
fit	if TRUE, the component models are fitted using get.models. See 'Details'.

Details

model.avg may be used either with a list of models, or directly with a model.selection object (e.g. returned by dredge). In the latter case, the models from the model selection table are not evaluated unless the argument fit is set to TRUE or some additional arguments are present (such as rank or dispersion). This results in much faster calculation, but has certain drawbacks, because

the fitted component model objects are not stored, and some methods (e.g. `predict`, `fitted`, `model.matrix` or `vcov`) would not be available with the returned object. Otherwise, `get.models` is called prior to averaging, and ... are passed to it.

For a list of model types that are accepted see list of supported models.

`rank` is found by a call to `match.fun` and typically is specified as a function or a symbol or a character string specifying a function to be searched for from the environment of the call to lapply. `rank` must be a function able to accept model as a first argument and must always return a numeric scalar.

Several standard methods for fitted model objects exist for class `averaging`, including `summary`, `predict`, `coef`, `confint`, `formula`, and `vcov`.

`coef`, `vcov`, `confint` and `coefTable` accept argument `full` that if set to `TRUE`, the full model-averaged coefficients are returned, rather than subset-averaged ones (when `full = FALSE`, being the default).

`logLik` returns a list of `logLik` objects for the component models.

Value

An object of class `"averaging"` is a list with components:

`msTable`	a `data.frame` with log-likelihood, *IC*, Δ_{IC} and 'Akaike weights' for the component models. Its attribute `"term.codes"` is a named vector with numerical representation of the terms in the row names of `msTable`.
`coefficients`	a `matrix` of model-averaged coefficients. "full" coefficients in first row, "subset" coefficients in second row. See 'Note'
`coefArray`	a 3-dimensional `array` of component models' coefficients, their standard errors and degrees of freedom.
`importance`	object of class `importance` containing relative importance values of each term (including interactions), calculated as a sum of the *Akaike weights* over all of the models in which the term appears.
`formula`	a formula corresponding to the one that would be used in a single model. The formula contains only the averaged (fixed) coefficients.
`call`	the matched call.

The object has following attributes:

`rank`	the rank function used.
`modelList`	optionally, a list of all component model objects. Only if the object was created with model objects (and not model selection table).
`beta`	Corresponds to the function argument.
`nobs`	number of observations.
`revised.var`	Corresponds to the function argument.

Note

The 'subset' (or 'conditional') average only averages over the models where the parameter appears. An alternative, the 'full' average assumes that a variable is included in every model, but in some models the corresponding coefficient (and its respective variance) is set to zero. Unlike the 'subset average', it does not have a tendency of biasing the value away from zero. The 'full' average is a type of shrinkage estimator and for variables with a weak relationship to the response they are smaller than 'subset' estimators.

Averaging models with different contrasts for the same factor would yield nonsense results, currently no checking for contrast consistency is done.

`print` method provides a concise output (similarly as for `lm`). To print more details use `summary` function, and `confint` to get confidence intervals.

Author(s)

Kamil Bartoń

References

Burnham, K. P. and Anderson, D. R. (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed. New York, Springer-Verlag.

Lukacs, P. M., Burnham K. P. and Anderson, D. R. (2009) *Model selection bias and Freedman's paradox*. Annals of the Institute of Statistical Mathematics 62(1): 117–125.

See Also

See `par.avg` for more details of model averaged parameter calculation.

`dredge`, `get.models`
`AICc` has examples of averaging models fitted by REML.

`modavg` in package **AICcmodavg**, and `coef.glmulti` in package **glmulti** also perform model averaging.

Examples

```
# Example from Burnham and Anderson (2002), page 100:
fm1 <- lm(y ~ ., data = Cement, na.action = na.fail)
(ms1 <- dredge(fm1))

#models with delta.aicc < 4
summary(model.avg(ms1, subset = delta < 4))

#or as a 95% confidence set:
avgmod.95p <- model.avg(ms1, cumsum(weight) <= .95)
confint(avgmod.95p)

## Not run:
# The same result, but re-fitting the models via 'get.models'
confset.95p <- get.models(ms1, cumsum(weight) <= .95)
model.avg(confset.95p)

# Force re-fitting the component models
model.avg(ms1, cumsum(weight) <= .95, fit = TRUE)
# Models are also fitted if additional arguments are given
model.avg(ms1, cumsum(weight) <= .95, rank = "AIC")

## End(Not run)

## Not run:
# using BIC (Schwarz's Bayesian criterion) to rank the models
BIC <- function(x) AIC(x, k = log(length(residuals(x))))
model.avg(confset.95p, rank = BIC)
```

```
# the same result, using AIC directly, with argument k
# 'x' in a quoted 'rank' argument is substituted with a model object
# (in this case it does not make much sense as the number of observations is
# common to all models)
model.avg(confset.95p, rank = AIC, rank.args = alist(k = log(length(residuals(x)))))

## End(Not run)
```

`model.sel` *model selection table*

Description

Build a model selection table.

Usage

```
model.sel(object, ...)

## Default S3 method:
model.sel(object, ..., rank = NULL, rank.args = NULL,
    beta = c("none", "sd", "partial.sd"), extra)
## S3 method for class 'model.selection'
model.sel(object, rank = NULL, rank.args = NULL, fit = NA,
    ..., beta = c("none", "sd", "partial.sd"), extra)
```

Arguments

`object`	a fitted model object, a list of such objects, or a `"model.selection"` object.
`...`	more fitted model objects.
`rank`	optional, custom rank function (returning an information criterion) to use instead of the default `AICc`, e.g. `QAIC` or `BIC`, may be omitted if `object` is a model list returned by `get.models`.
`rank.args`	optional `list` of arguments for the `rank` function. If one is an expression, an `x` within it is substituted with a current model.
`fit`	logical, stating whether the model objects should be re-fitted if they are not stored in the `"model.selection"` object. Set to `NA` to re-fit the models only if this is needed. See 'Details'.
`beta`	indicates whether and how the component models' coefficients should be standardized. See the argument's description in `dredge`.
`extra`	optional additional statistics to include in the result, provided as functions, function names or a list of such (best if named or quoted). See `dredge` for details.

Details

`model.sel` used with `"model.selection"` object will re-fit model objects, unless they are stored in `object` (in attribute `"modelList"`), if argument `extra` is provided, or the requested `beta` is different than object's `"beta"` attribute, or the new `rank` function cannot be applied directly to `logLik` objects, or new `rank.args` are given (unless argument `fit = FALSE`).

Value

An object of class `c("model.selection", "data.frame")`, being a `data.frame`, where each row represents one model and columns contain useful information about each model: the coefficients, *df*, log-likelihood, the value of the information criterion used, Δ_{IC} and 'Akaike weight'. If any arguments differ between the modelling function calls, the result will include additional columns showing them (except for formulas and some other arguments).

See `model.selection.object` for its structure.

Author(s)

Kamil Bartoń

See Also

`dredge`, `AICc`, list of supported models.

Possible alternatives: `ICtab` (in package **bbmle**), or `aictab` (**AICcmodavg**).

Examples

```
Cement$X1 <- cut(Cement$X1, 3)
Cement$X2 <- cut(Cement$X2, 2)

fm1 <- glm(formula = y ~ X1 + X2 * X3, data = Cement)
fm2 <- update(fm1, . ~ . - X1 - X2)
fm3 <- update(fm1, . ~ . - X2 - X3)

## ranked with AICc by default
(msAICc <- model.sel(fm1, fm2, fm3))

## ranked with BIC
model.sel(fm1, fm2, fm3, rank = AIC, rank.args = alist(k = log(nobs(x))))
# or
# model.sel(msAICc, rank = AIC, rank.args = alist(k = log(nobs(x))))
# or
# update(msAICc, rank = AIC, rank.args = alist(k = log(nobs(x))))
```

`model.selection.object`

Description of Model Selection Objects

Description

An object of class `"model.selection"` holds a table of model coefficients and ranking statistics. It is a produced by `dredge` or `model.sel`.

Value

The object is a `data.frame` with additional attributes. Each row represents one model. The models are ordered by the information criterion value specified by `rank` (lowest on top).

Data frame columns:

`model terms`	For numeric covariates these columns hold coeficent value, for factors their presence in the model. If the term is not present in a model, value is `NA`.
`'varying' arguments`	optional. If any arguments differ between the modelling function calls (except for formulas and some other arguments), these will be held in additional columns (of class `"factor"`).
`"df"`	Number of model parameters
`"logLik"`	Log-likelihood (or quasi-likelihood for GEE)
`rank`	Information criterion value
`"delta"`	Δ_{IC}
`"weight"`	'Akaike weights'.

Attributes:

`model.calls`	A list containing model calls (arranged in the same order as in the table). A model call can be retrieved with `getCall(*, i)` where i is a vector of model index or name (if given as character string).
`global`	The `global.model` object
`global.call`	Call to the `global.model`
`terms`	A character string holding all term names. Attribute `"interceptLabel"` gives the name of intercept term.
`rank`	The `rank` function used
`beta`	A character string, representing the coefficient standardizing method used. Either `"none"`, `"sd"` or `"partial.sd"`
`coefTables`	List of matrices of class `"coefTable"` containing each model's coefficents with std. errors and associated dfs
`nobs`	Number of observations
`warnings`	optional (`pdredge` only). A list of errors and warnings issued by the modelling function during the fitting, with model number appended to each.

Most attributes does not need (and should not) be accessed directly, use of extractor functions is preferred. These functions include `getCall` for retrieving model calls, `coefTable` and `coef` for coefficiens, and `nobs`. `logLik` extracts list of model log-likelihoods (as `"logLik"` objects), and `Weights` extracts 'Akaike weights'.

The object has class `c("model.selection", "data.frame")`.

See Also

`dredge`, `model.sel`.

MuMIn-models *List of supported models*

Description

List of model classes accepted by `model.avg`, `model.sel`, and `dredge`.

Details

Fitted model objects that can be used with model selection and model averaging functions include those produced by:

- `lm`, `glm` (package **stats**);
- `rlm`, `glm.nb` and `polr` (**MASS**);
- `multinom` (**nnet**);
- `lme`, `gls` (**nlme**);
- `lmer`, `glmer` (**lme4**);
- `cpglm`, `cpglmm` (**cplm**);
- `gam`, `gamm*` (**mgcv**);
- `gamm4*` (**gamm4**);
- `glmmML` (**glmmML**);
- `glmmadmb` (**glmmADMB** from R-Forge);
- `glmmTMB` (**glmmTMB**);
- `MCMCglmm*` (**MCMCglmm**);
- `asreml` (non-free commercial package **asreml**; allows only for REML comparisons);
- `hurdle`, `zeroinfl` (**pscl**);
- `negbin`, `betabin` (class `"glimML"`), package **aod**);
- `aodml`, `aodql` (**aods3**);
- `betareg` (**betareg**);
- `brglm` (**brglm**);
- `*sarlm` models, `spautolm` (**spdep**);
- `spml*` (if fitted by ML, **splm**);
- `coxph`, `survreg` (**survival**);
- `coxme`, `lmekin` (**coxme**);
- `rq` (**quantreg**);
- `clm` and `clmm` (**ordinal**);
- `logistf` (**logistf**);
- `crunch*`, `pgls` (**caper**);
- `maxlike` (**maxlike**);
- functions from package **unmarked** (within the class `"unmarkedFit"`);
- `mark` and related functions (class `mark` from package **RMark**). Currently `dredge` can only manipulate `formula` element of the argument `model.parameters`, keeping its other elements intact.

Generalized Estimation Equation model implementations: geeglm from package **geepack**, gee from **gee**, geem from **geeM**, and yags from **yags** (from R-Forge) can be used with QIC as the selection criterion.

Other classes are also likely to be supported, in particular if they inherit from one of the above classes. In general, the models averaged with model.avg may belong to different types (e.g. glm and gam), provided they use the same data and response, and if it is valid to do so. This applies also to constructing model selection tables with model.sel.

Note

* In order to use gamm, gamm4, spml (> 1.0.0), crunch or MCMCglmm with dredge, an updateable wrapper for these functions should be created.

See Also

model.avg, model.sel and dredge.

---

nested *Identify nested models*

Description

Find models that are 'nested' within each model in the model selection table.

Usage

```
nested(x, indices = c("none", "numeric", "rownames"), rank = NULL)
```

Arguments

x	a "model.selection" object (result of dredge or model.sel).
indices	if omitted or "none" then the function checks if, for each model, there are any higher ranked models nested within it. If "numeric" or "rownames", indices or names of all nested models are returned. See "Value".
rank	the name of the column with the ranking values (defaults to the one before "delta"). Only used if indices is "none".

Details

In model comparison, a model is said to be "nested" within another model if it contains a subset of parameters of the latter model, but does not include other parameters (e.g. model 'A+B' is nested within 'A+B+C' but not 'A+C+D').

This function can be useful in a model selection approach suggested by Richards (2008), in which more complex variants of any model with a lower IC value are excluded from the candidate set.

Value

A vector of length equal to the number of models (table rows).

If `indices = "none"` (the default), it is a vector of logical values where *i*-th element is `TRUE` if any model(s) higher up in the table are nested within it (i.e. if simpler models have lower IC pointed by `rank`).

For `indices` other than `"none"`, the function returns a list of vectors of numeric indices or names of models nested within each *i*-th model.

Note

This function determines nesting based only on fixed model terms, within groups of models sharing the same 'varying' parameters (see `dredge` and example in `Beetle`).

Author(s)

Kamil Bartoń

References

Richards, S. A., Whittingham, M. J., Stephens, P. A (2011). Model selection and model averaging in behavioural ecology: the utility of the IT-AIC framework. *Behavioral Ecology and Sociobiology*, 65: 77-89

Richards, S. A (2008) Dealing with overdispersed count data in applied ecology. *Journal of Applied Ecology* 45: 218–227

See Also

`dredge`, `model.sel`

Examples

```
fm <- lm(y ~ X1 + X2 + X3 + X4, data = Cement, na.action = na.fail)
ms <- dredge(fm)

# filter out overly complex models according to the
# "nesting selection rule":
subset(ms, !nested(.)) # dot represents the ms table object

# print model "4" and all models nested within it
nst <- nested(ms, indices = "row")
ms[c("4", nst[["4"]])]

ms$nested <- sapply(nst, paste, collapse = ",")

ms
```

`par.avg` *Parameter averaging*

Description

Average a coefficient with standard errors based on provided weights. This function is intended chiefly for internal use.

Usage

```
par.avg(x, se, weight, df = NULL, level = 1 - alpha, alpha = 0.05,
    revised.var = TRUE, adjusted = TRUE)
```

Arguments

`x`	vector of parameters.
`se`	vector of standard errors.
`weight`	vector of weights.
`df`	optional vector of degrees of freedom.
`alpha, level`	significance level for calculating confidence intervals.
`revised.var`	logical, should the revised formula for standard errors be used? See 'Details'.
`adjusted`	logical, should the inflated standard errors be calculated? See 'Details'.

Details

Unconditional standard errors are square root of the variance estimator, calculated either according to the original equation in Burnham and Anderson (2002, equation 4.7), or a newer, revised formula from Burnham and Anderson (2004, equation 4) (if `revised.var = TRUE`, this is the default). If `adjusted = TRUE` (the default) and degrees of freedom are given, the adjusted standard error estimator and confidence intervals with improved coverage are returned (see Burnham and Anderson 2002, section 4.3.3).

Value

`par.avg` returns a vector with named elements:

`Coefficient`	model coefficients
`SE`	unconditional standard error
`Adjusted SE`	adjusted standard error
`Lower CI, Upper CI`	unconditional confidence intervals.

Author(s)

Kamil Bartoń

References

Burnham, K. P. and Anderson, D. R. (2002) *Model selection and multimodel inference: a practical information-theoretic approach*. 2nd ed.

Burnham, K. P. and Anderson, D. R. (2004) *Multimodel inference - understanding AIC and BIC in model selection*. Sociological Methods & Research 33(2): 261-304.

See Also

`model.avg` for model averaging.

`pdredge` *Automated model selection using parallel computation*

Description

Parallelized version of `dredge`.

Usage

```
pdredge(global.model, cluster = NA,
    beta = c("none", "sd", "partial.sd"), evaluate = TRUE, rank = "AICc",
    fixed = NULL, m.lim = NULL, m.min, m.max, subset, trace = FALSE,
    varying, extra, ct.args = NULL, check = FALSE, ...)
```

Arguments

`global.model, beta, evaluate, rank`	see `dredge`.
`fixed, m.lim, m.max, m.min, subset, varying, extra, ct.args, ...`	see `dredge`.
`trace`	displays the generated calls, but may not work as expected since the models are evaluated in batches rather than one by one.
`cluster`	either a valid `"cluster"` object, or `NA` for a single threaded execution.
`check`	either integer or logical value controlling how much checking for existence and correctness of dependencies is done on the cluster nodes. See 'Details'.

Details

All the dependencies for fitting the `global.model`, including the data and any objects the modelling function will use must be exported into the cluster worker nodes (e.g. *via* `clusterExport`). The required packages must be also loaded thereinto (e.g. *via* `clusterEvalQ(..., library(package))`, before the cluster is used by `pdredge`.

If `check` is `TRUE` or positive, `pdredge` tries to check whether all the variables and functions used in the call to `global.model` are present in the cluster nodes' `.GlobalEnv` before proceeding further. This causes false errors if some arguments of the model call (other than `subset`) would be evaluated in `data` environment. In that case using `check = FALSE` (the default) is desirable.

If `check` is `TRUE` or greater than one, `pdredge` will compare the `global.model` updated at the cluster nodes with the one given as argument.

Value

See dredge.

Author(s)

Kamil Bartoń

See Also

makeCluster and other cluster related functions in packages **parallel** or **snow**.

Examples

```
# One of these packages is required:
## Not run: require(parallel) || require(snow)

# From example(Beetle)

Beetle100 <- Beetle[sample(nrow(Beetle), 100, replace = TRUE),]

fm1 <- glm(Prop ~ dose + I(dose^2) + log(dose) + I(log(dose)^2),
    data = Beetle100, family = binomial, na.action = na.fail)

msubset <- expression(xor(dose, `log(dose)`) & (dose | !`I(dose^2)`)
    & (`log(dose)` | !`I(log(dose)^2)`))
varying.link <- list(family = alist(logit = binomial("logit"),
    probit = binomial("probit"), cloglog = binomial("cloglog") ))

# Set up the cluster
clusterType <- if(length(find.package("snow", quiet = TRUE))) "SOCK" else "PSOCK"
clust <- try(makeCluster(getOption("cl.cores", 2), type = clusterType))

clusterExport(clust, "Beetle100")

# noticeable gain only when data has about 3000 rows (Windows 2-core machine)
print(system.time(dredge(fm1, subset = msubset, varying = varying.link)))
print(system.time(pdredge(fm1, cluster = FALSE, subset = msubset,
    varying = varying.link)))
print(system.time(pdd <- pdredge(fm1, cluster = clust, subset = msubset,
    varying = varying.link)))

print(pdd)

## Not run:
# Time consuming example with 'unmarked' model, based on example(pcount).
# Having enough patience you can run this with 'demo(pdredge.pcount)'.
library(unmarked)
data(mallard)
mallardUMF <- unmarkedFramePCount(mallard.y, siteCovs = mallard.site,
    obsCovs = mallard.obs)
(ufm.mallard <- pcount(~ ivel + date + I(date^2) ~ length + elev + forest,
    mallardUMF, K = 30))
clusterEvalQ(clust, library(unmarked))
```

```
clusterExport(clust, "mallardUMF")

# 'stats4' is needed for AIC to work with unmarkedFit objects but is not
# loaded automatically with 'unmarked'.
require(stats4)
invisible(clusterCall(clust, "library", "stats4", character.only = TRUE))

#system.time(print(pdd1 <- pdredge(ufm.mallard,
#   subset = `p(date)` | !`p(I(date^2))`, rank = AIC)))

system.time(print(pdd2 <- pdredge(ufm.mallard, clust,
    subset = `p(date)` | !`p(I(date^2))`, rank = AIC, extra = "adjR^2")))


# best models and null model
subset(pdd2, delta < 2 | df == min(df))

# Compare with the model selection table from unmarked
# the statistics should be identical:
models <- get.models(pdd2, delta < 2 | df == min(df), cluster = clust)

modSel(fitList(fits = structure(models, names = model.names(models,
    labels = getAllTerms(ufm.mallard)))), nullmod = "(Null)")

## End(Not run)

stopCluster(clust)
```

`plot.model.selection` *Visualize model selection table*

Description

Produces a graphical representation of model weights and relative variable importance.

Usage

```
## S3 method for class 'model.selection'
plot(x, ylab = NULL, xlab = NULL,
    labels = attr(x, "terms"), labAsExpr = FALSE,
    col = c("SlateGray", "SlateGray2"), col2 = "white", border = par("col"),
    par.lab = NULL, par.vlab = NULL,
    axes = TRUE, ann = TRUE, ...)
```

Arguments

`x`	a `"model.selection"` object.
`xlab, ylab`	labels for the x and y axis.
`labels`	optional, a character vector or an expression containing model term labels (to appear on top side of the plot). Its length must be equal to number of model terms in the table. Defaults to model term names.

labAsExpr	a logical indicating whether the character labels should be interpreted (parsed) as R expressions.
col, col2	vector of colors for columns (if more than one col is given, columns will be filled with alternating colors). If col2 is specified cells will be filled with gradient from col to col2. Set col2 to NA for no gradient.
border	border color for cells and axes.
par.lab, par.vlab	optional lists or parameters for term labels (top axis) and model names (right axis), respectively.
axes, ann	logical values indicating whether the axis and annotation should appear on the plot.
...	further graphical parameters to be set for the plot (see par).

Author(s)

Kamil Bartoń

See Also

plot.default, par

For examples, see 'MuMIn-package'

---

predict.averaging *Predict method for averaged models*

Description

Model-averaged predictions, optionally with standard errors.

Usage

```
## S3 method for class 'averaging'
predict(object, newdata = NULL, se.fit = FALSE,
    interval = NULL, type = NA, backtransform = FALSE, full = TRUE, ...)
```

Arguments

object	an object returned by model.avg.
newdata	optional data.frame in which to look for variables with which to predict. If omitted, the fitted values are used.
se.fit	logical, indicates if standard errors should be returned. This has any effect only if the predict methods for each of the component models support it.
interval	currently not used.
type	the type of predictions to return (see documentation for predict appropriate for the class of used component models). If omitted, the default type is used. See 'Details'.

backtransform	if TRUE, the averaged predictions are back-transformed from link scale to response scale. This makes sense provided that all component models use the same family, and the prediction from each of the component models is calculated on the link scale (as specified by type. For glm, use type = "link"). See 'Details'.
full	if TRUE, the full model averaged coefficients are used (only if se.fit = FALSE and the component objects are a result of lm).
...	arguments to be passed to respective predict method (e.g. level for lme model).

Details

predicting is possible only with averaging objects with "modelList" attribute, i.e. those created *via* model.avg from a model list, or from model.selection object with argument fit = TRUE (which will recreate the model objects, see model.avg).

If all the component models are oridinary linear models, the prediction can be made either with the full averaged coefficients (the argument full = TRUE this is the default) or subset-averaged coefficients. Otherwise the prediction is obtained by calling predict on each component model and weighted averaging the results, which corresponds to the assumption that all predictors are present in all models, but those not estimated are equal zero (see 'Note' in model.avg). Predictions from component models with standard errors are passed to par.avg and averaged in the same way as the coefficients are.

Predictions on the response scale from generalized models can be calculated by averaging predictions of each model on the link scale, followed by inverse transformation (this is achieved with type = "link" and backtransform = TRUE). This is only possible if all component models use the same family and link function. Alternatively, predictions from each model on response scale may be averaged (with type = "response" and backtransform = FALSE). Note that this leads to results differing from those calculated with the former method. See also predict.glm.

Value

If se.fit = FALSE, a vector of predictions, otherwise a list with components: fit containing the predictions, and se.fit with the estimated standard errors.

Note

This method relies on availability of the predict methods for the component model classes (except when all component models are of class lm).

The package **MuMIn** includes predict methods for lme, gls and lmer (**lme4**), all of which can calculate standard errors of the predictions (with se.fit = TRUE). The former two enhance the original predict methods from package **nlme**, and with se.fit = FALSE they return identical result. **MuMIn**'s versions are always used in averaged model predictions (so it is possible to predict with standard errors), but from within global environment they will be found only if **MuMIn** is before **nlme** on the search list (or directly extracted from namespace as MuMIn:::predict.lme).

predict method for mer models currently can only calculate values on the outermost level (equivalent to level = 0 in predict.lme).

Author(s)

Kamil Bartoń

See Also

model.avg, and par.avg for details of model-averaged parameter calculation.

predict.lme, predict.gls

Examples

```
# Example from Burnham and Anderson (2002), page 100:
fm1 <- lm(y ~ X1 + X2 + X3 + X4, data = Cement)

ms1 <- dredge(fm1)
confset.95p <- get.models(ms1, subset = cumsum(weight) <= .95)
avgm <- model.avg(confset.95p)

nseq <- function(x, len = length(x)) seq(min(x, na.rm = TRUE),
    max(x, na.rm=TRUE), length = len)

# New predictors: X1 along the range of original data, other
# variables held constant at their means
newdata <- as.data.frame(lapply(lapply(Cement[, -1], mean), rep, 25))
newdata$X1 <- nseq(Cement$X1, nrow(newdata))

n <- length(confset.95p)

# Predictions from each of the models in a set, and with averaged coefficients
pred <- data.frame(
model = sapply(confset.95p, predict, newdata = newdata),
averaged.subset = predict(avgm, newdata, full = FALSE),
    averaged.full = predict(avgm, newdata, full = TRUE)
)

opal <- palette(c(topo.colors(n), "black", "red", "orange"))
matplot(newdata$X1, pred, type = "l",
lwd = c(rep(2,n),3,3), lty = 1,
    xlab = "X1", ylab = "y", col=1:7)

# For comparison, prediction obtained by averaging predictions of the component
# models
pred.se <- predict(avgm, newdata, se.fit = TRUE)
y <- pred.se$fit
ci <- pred.se$se.fit  * 2
matplot(newdata$X1, cbind(y, y - ci, y + ci), add = TRUE, type="l",
lty = 2, col = n + 3, lwd = 3)

legend("topleft",
    legend=c(lapply(confset.95p, formula),
        paste(c("subset", "full"), "averaged"), "averaged predictions + CI"),
    lty = 1, lwd = c(rep(2,n),3,3,3),  cex = .75, col=1:8)

palette(opal)
```

QAIC *Quasi AIC or AICc*

Description

Calculate a modification of Akaike's Information Criterion for overdispersed count data (or its version corrected for small sample, "quasi-AIC$_c$"), for one or several fitted model objects.

Usage

```
QAIC(object, ..., chat, k = 2, REML = NULL)
QAICc(object, ..., chat, k = 2, REML = NULL)
```

Arguments

`object`	a fitted model object.
`...`	optionally, more fitted model objects.
`chat`	$\hat{c}$, the variance inflation factor.
`k`	the 'penalty' per parameter.
`REML`	optional logical value, passed to the `logLik` method indicating whether the restricted log-likelihood or log-likelihood should be used. The default is to use the method used for model estimation.

Value

If only one object is provided, returns a numeric value with the corresponding QAIC or QAIC$_c$; otherwise returns a `data.frame` with rows corresponding to the objects.

Note

$\hat{c}$ is the dispersion parameter estimated from the global model, and can be calculated by dividing model's deviance by the number of residual degrees of freedom.

In calculation of QAIC, the number of model parameters is increased by 1 to account for estimating the overdispersion parameter. Without overdispersion, $\hat{c} = 1$ and QAIC is equal to AIC.

Note that `glm` does not compute maximum-likelihood estimates in models within the *quasi-* family. In case it is justified, it can be worked around by 'borrowing' the `aic` element from the corresponding 'non-quasi' family (see 'Example').

Author(s)

Kamil Bartoń

See Also

`AICc`, `quasi` family used for models with over-dispersion

Examples

```
options(na.action = "na.fail")

# Based on "example(predict.glm)", with one number changed to create
# overdispersion
budworm <- data.frame(
    ldose = rep(0:5, 2), sex = factor(rep(c("M", "F"), c(6, 6))),
    numdead = c(10, 4, 9, 12, 18, 20, 0, 2, 6, 10, 12, 16))
budworm$SF = cbind(numdead = budworm$numdead,
    numalive = 20 - budworm$numdead)

budworm.lg <- glm(SF ~ sex*ldose, data = budworm, family = binomial)
(chat <- deviance(budworm.lg) / df.residual(budworm.lg))

dredge(budworm.lg, rank = "QAIC", chat = chat)
dredge(budworm.lg, rank = "AIC")



## Not run:
# A 'hacked' constructor for quasibinomial family object that allows for
# ML estimation
hacked.quasibinomial <- function(...) {
    res <- quasibinomial(...)
    res$aic <- binomial(...)$aic
    res
}
QAIC(update(budworm.lg, family = hacked.quasibinomial), chat = chat)

## End(Not run)
```

QIC *QIC and quasi-Likelihood for GEE*

Description

Calculate quasi-likelihood under the independence model criterion (QIC) for Generalized Estimating Equations.

Usage

```
QIC(object, ..., typeR = FALSE)
QICu(object, ..., typeR = FALSE)
quasiLik(object, ...)
```

Arguments

object	a fitted model object of class "gee", "geepack", "geem" or "yags".
...	for QIC and QIC$_u$, optionally more fitted model objects.
typeR	logical, whether to calculate QIC(R). QIC(R) is based on quasi-likelihood of a working correlation R model. Defaults to FALSE, and QIC(I) based on independence model is returned.

Value

If just one object is provided, returns a numeric value with the corresponding QIC; if more than one object are provided, returns a `data.frame` with rows corresponding to the objects and one column representing QIC or QIC_u.

Note

This implementation is based partly on (revised) code from packages **yags** (R-Forge) and **ape**.

Author(s)

Kamil Bartoń

References

Pan W. (2001) Akaike's Information Criterion in Generalized Estimating Equations. *Biometrics* 57: 120-125

Hardin J. W., Hilbe, J. M. (2003) *Generalized Estimating Equations*. Chapman & Hall/CRC

See Also

Methods exist for `gee` (package **gee**), `geeglm` (**geepack**), `geem` (**geeM**), and `yags` (**yags** on R-Forge). `yags` and `compar.gee` from package **ape** both provide QIC values.

Examples

```
data(ohio)

fm1 <- geeglm(resp ~ age * smoke, id = id, data = ohio,
    family = binomial, corstr = "exchangeable", scale.fix = TRUE)
fm2 <- update(fm1, corstr = "ar1")
fm3 <- update(fm1, corstr = "unstructured")

model.sel(fm1, fm2, fm3, rank = QIC)

## Not run:
# same result:
    dredge(fm1, m.lim = c(3, NA), rank = QIC, varying = list(
    corstr = list("exchangeable", "unstructured", "ar1")
    ))

## End(Not run)
```

`r.squaredGLMM` *Pseudo-R-squared for Generalized Mixed-Effect models*

Description

Calculate conditional and marginal coefficient of determination for Generalized mixed-effect models (R^2_{GLMM}).

Usage

```
r.squaredGLMM(x)
```

Arguments

`x`	a fitted linear model object.

Details

For mixed-effects models, R^2 can be categorized into two types. **Marginal** R^2_{GLMM} represents the variance explained by fixed factors, and is defined as:

$$R^2_{GLMM(m)} = \frac{\sigma^2_f}{\sigma^2_f + \sum_{l=1}^{u} \sigma^2_l + \sigma^2_e + \sigma^2_d}$$

Conditional R^2_{GLMM} is interpreted as variance explained by both fixed and random factors (i.e. the entire model), and is calculated according to the equation:

$$R^2_{GLMM(c)} = \frac{\sigma^2_f + \sum_{l=1}^{u} \sigma^2_l}{\sigma^2_f + \sum_{l=1}^{u} \sigma^2_l + \sigma^2_e + \sigma^2_d}$$

where σ^2_f is the variance of the fixed effect components, and $\sum \sigma^2_l$ is the sum of all u variance components (group, individual, etc.), σ^2_e is the variance due to additive dispersion and σ^2_d is the distribution-specific variance.

Value

`r.squaredGLMM` returns a numeric vector with two values for marginal and conditional R^2_{GLMM}.

Note

R^2_{GLMM} can be calculated also for fixed-effect models. In the simpliest case of OLS it reduces to `var(fitted) / (var(fitted) + deviance / 2)`. Unlike likelihood-ratio based R^2 for OLS, value of this statistic differs from that of the classical R^2.

Currently methods exist for classes: `mer`(`Mod`), `lme`, `glmmML` and (`g`)`lm`.

See note in `r.squaredLR` help page for comment on using R^2 in model selection.

Author(s)

This implementation is based on R code from 'Supporting Information' for Nakagawa & Schielzeth (2012), and its extension by Paul Johnson.

References

Nakagawa, S, Schielzeth, H. (2013). A general and simple method for obtaining R^2 from Generalized Linear Mixed-effects Models. *Methods in Ecology and Evolution* 4: 133–142

Johnson, P.C.D. (2014) Extension Nakagawa & Schielzeth's R^2_{GLMM} to random slopes models. *Methods in Ecology and Evolution* 5: 44-946.

See Also

summary.lm, r.squaredLR

Examples

```
data(Orthodont, package = "nlme")

fm1 <- lme(distance ~ Sex * age, ~ 1 | Subject, data = Orthodont)

r.squaredGLMM(fm1)
r.squaredLR(fm1)
r.squaredLR(fm1, null.RE = TRUE)
```

---

r.squaredLR *Likelihood-ratio based pseudo-R-squared*

Description

Calculate a coefficient of determination based on the likelihood-ratio test (R^2_{LR}).

Usage

```
r.squaredLR(x, null = NULL, null.RE = FALSE)

null.fit(x, evaluate = FALSE, RE.keep = FALSE, envir = NULL)
```

Arguments

x	a fitted model object.
null	a fitted *null* model. If not provided, null.fit will be used to construct it. null.fit's capabilities are limited to only a few model classes, for others the *null* model has to be specified manually.
null.RE	logical, should the null model contain random factors? Only used if no *null* model is given, otherwise omitted, with a warning.
evaluate	if TRUE evaluate the fitted model object else return the call.
RE.keep	if TRUE, the random effects of the original model are included.
envir	the environment in which the *null* model is to be evaluated, defaults to the environment of the original model's formula.

Details

This statistic is is one of the several proposed pseudo-R^2's for nonlinear regression models. It is based on an improvement from *null* (intercept only) model to the fitted model, and calculated as

$$R^2_{LR} = 1 - \exp(-\frac{2}{n}(\log \mathcal{L}(x) - \log \mathcal{L}(0)))$$

where $\log \mathcal{L}(x)$ and $\log \mathcal{L}(0)$ are the log-likelihoods of the fitted and the *null* model respectively. ML estimates are used if models have been fitted by REstricted ML (by calling `logLik` with argument `REML = FALSE`). Note that the *null* model can include the random factors of the original model, in which case the statistic represents the 'variance explained' by fixed effects.

For OLS models the value is consistent with classical R^2. In some cases (e.g. in logistic regression), the maximum R^2_{LR} is less than one. The modification proposed by Nagelkerke (1991) adjusts the R^2_{LR} to achieve 1 at its maximum: $\bar{R}^2 = R^2_{LR}/\max(R^2_{LR})$ where $\max(R^2_{LR}) = 1 - \exp(\frac{2}{n}\log \mathcal{L}(0))$.

`null.fit` tries to guess the *null* model call, given the provided fitted model object. This would be usually a `glm`. The function will give an error for an unrecognized class.

Value

`r.squaredLR` returns a value of R^2_{LR}, and the attribute `"adj.r.squared"` gives the Nagelkerke's modified statistic. Note that this is not the same as nor equivalent to the classical 'adjusted R squared'.

`null.fit` returns the fitted *null* model object (if `evaluate = TRUE`) or an unevaluated call to fit a *null* model.

Note

R^2 is a useful goodness-of-fit measure as it has the interpretation of the proportion of the variance 'explained', but it performs poorly in model selection, and is not suitable for use in the same way as the information criterions.

References

Cox, D. R. and Snell, E. J. (1989) *The analysis of binary data*, 2nd ed. London, Chapman and Hall

Magee, L. (1990) R^2 measures based on Wald and likelihood ratio joint significance tests. *Amer. Stat.* 44: 250-253

Nagelkerke, N. J. D. (1991) A note on a general definition of the coefficient of determination. *Biometrika* 78: 691-692

See Also

`summary.lm`, `r.squaredGLMM`

`stackingWeights` *Stacking model weights*

Description

Computes model weights based on a cross-validation-like procedure.

Usage

```
stackingWeights(object, ..., data, R, p = 0.5)
```

Arguments

`object, ...`	two or more fitted `glm` objects, or a `list` of such, or an `"averaging"` object.
`data`	a data frame containing the variables in the model, used for fitting and prediction.
`R`	the number of replicates.
`p`	the proportion of the `data` to be used as training set. Defaults to 0.5.

Details

Each model in a set is fitted to the training data: a subset of `p * N` observations in `data`. From these models a prediction is produced on the remaining part of `data` (the test or hold-out data). These hold-out predictions are fitted to the hold-out observations, by optimising the weights by which the models are combined. This process is repeated `R` times, yielding a distribution of weights for each model (which Smyth & Wolpert (1998) referred to as an 'empirical Bayesian estimate of posterior model probability'). A mean or median of model weights for each model is taken and re-scaled to sum to one.

Value

`stackingWeights` returns a matrix with two rows, holding model weights calculated using `mean` and `median`.

Note

This approach requires a sample size of at least $2\times$ the number of models.

Author(s)

Carsten Dormann, Kamil Bartoń

References

Wolpert, D. H. (1992) Stacked generalization. *Neural Networks*, 5: 241-259.

Smyth, P. & Wolpert, D. (1998) *An Evaluation of Linearly Combining Density Estimators via Stacking. Technical Report No. 98-25.* Information and Computer Science Department, University of California, Irvine, CA.

See Also

Weights, model.avg

Other model.weights: BGWeights, bootWeights, cos2Weights, jackknifeWeights

Examples

```
# global model fitted to training data:
fm <- glm(y ~ X1 + X2 + X3 + X4, data = Cement, na.action = na.fail)
# generate a list of *some* subsets of the global model
models <- lapply(dredge(fm, evaluate = FALSE, fixed = "X1", m.lim = c(1, 3)), eval)

wts <- stackingWeights(models, data = Cement, R = 10)

ma <- model.avg(models)
Weights(ma) <- wts["mean", ]

predict(ma)
```

---

std.coef *Standardized model coefficients*

Description

Standardize model coefficients by Standard Deviation or Partial Standard Deviation.

Usage

```
std.coef(x, partial.sd, ...)

partial.sd(x)

# Deprecated:
beta.weights(model)
```

Arguments

x, model	a fitted model object.
partial.sd	logical, if set to TRUE, model coefficients are multiplied by partial SD, otherwise they are multiplied by the ratio of the standard deviations of the independent variable and dependent variable.
...	additional arguments passed to coefTable, e.g. dispersion.

Details

Standardizing model coefficients has the same effect as centring and scaling the input variables. "Classical" standardized coefficients are calculated as $\beta_i^* = \beta_i \frac{s_{X_i}}{s_y}$, where β is the unstandardized coefficient, s_{X_i} is the standard deviation of associated depenent variable X_i and s_y is SD of the response variable.

If the variables are intercorrelated, the standard deviation of X_i used in computing the standardized coefficients β_i^* should be replaced by a partial standard deviation of X_i which is adjusted for the multiple correlation of X_i with the other X variables included in the regression equation. The partial standard deviation is calculated as $s^*_{X_i} = s_{X_i} VIF(X_i)^{-0.5}(\frac{n-1}{n-p})^{0.5}$, where *VIF* is the variance inflation factor, *n* is the number of observations and *p* number of predictors in the model. Coefficient is then transformed as $\beta_i^* = \beta_i s^*_{X_i}$.

Value

A matrix with at least two columns for standardized coefficient estimate and its standard error. Optionally, third column holds degrees of freedom associated with the coefficients.

Author(s)

Kamil Bartoń. Variance inflation factors calculation is based on function `vif` from package **car** written by Henric Nilsson and John Fox.

References

Cade, B.S. (2015) Model averaging and muddled multimodel inferences. *Ecology* 96, 2370-2382.

Afifi A., May S., Clark V.A. (2011) *Practical Multivariate Analysis*, Fifth Edition. CRC Press.

Bring, J. (1994). How to standardize regression coefficients. *The American Statistician* 48, 209-213.

See Also

`partial.sd` can be used with `stdize`.

`coef` or `coeffs` and `coefTable` for unstandardized coefficients.

Examples

```
# Fit model to original data:
fm  <- lm(y ~ x1 + x2 + x3 + x4, data = GPA)

# Partial SD for the default formula:   y ~ x1 + x2 + x3 + x4
psd <- partial.sd(lm(data = GPA))[-1] # remove first element for intercept

# Standardize data:
zGPA <- stdize(GPA, scale = c(NA, psd), center = TRUE)
# Note: first element of 'scale' is set to NA to ignore the first column 'y'

# Coefficients of a model fitted to standardized data:
zapsmall(coefTable(stdizeFit(fm, data = zGPA)))
# Standardized coefficients of a model fitted to original data:
zapsmall(std.coef(fm, partial = TRUE))


# Standardizing nonlinear models:
fam <- Gamma("inverse")
fmg <- glm(log(y) ~ x1 + x2 + x3 + x4, data = GPA, family = fam)

psdg <- partial.sd(fmg)
zGPA <- stdize(GPA, scale = c(NA, psdg[-1]), center = FALSE)
fmgz <- glm(log(y) ~ z.x1 + z.x2 + z.x3 + z.x4, zGPA, family = fam)
```

```
# Coefficients using standardized data:
coef(fmgz) # (intercept is unchanged because the variables haven't been
           #  centred)
# Standardized coefficients:
coef(fmg) * psdg
```

`stdize` *Standardize data*

Description

`stdize` standardizes variables by centring and scaling.

`stdizeFit` modifies a model call or existing model to use standardized variables.

Usage

```
## Default S3 method:
stdize(x, center = TRUE, scale = TRUE, ...)

## S3 method for class 'logical'
stdize(x, binary = c("center", "scale", "binary", "half", "omit"),
    center = TRUE, scale = FALSE, ...)
## also for two-level factors

## S3 method for class 'data.frame'
stdize(x, binary = c("center", "scale", "binary", "half", "omit"),
    center = TRUE, scale = TRUE, omit.cols = NULL, source = NULL,
    prefix = TRUE, append = FALSE, ...)

## S3 method for class 'formula'
stdize(x, data = NULL, response = FALSE,
    binary = c("center", "scale", "binary", "half", "omit"),
    center = TRUE, scale = TRUE, omit.cols = NULL, prefix = TRUE,
    append = FALSE, ...)

stdizeFit(object, data, which = c("formula", "subset", "offset", "weights"),
    evaluate = TRUE, quote = NA)
```

Arguments

`x`	a numeric or logical vector, factor, numeric matrix, `data.frame` or a formula.
`center, scale`	either a logical value, or a logical or numeric vector of length equal to the number of columns of `x` (see 'Details'). `scale` can be also a function to use for scaling.
`binary`	specifies how binary variables (logical or two-level factors) are scaled. Default is to `"center"` by subtracting the mean assuming levels are equal to 0 and 1; use `"scale"` to both centre and scale by SD, `"binary"` to centre to 0 / 1, `"half"` to centre to -0.5 / 0.5, and `"omit"` to leave binary variables unmodified. This argument has precedence over `center` and `scale`, unless it is set to `NA` (in which case binary variables are treated like numeric variables).

source	a reference data.frame, being a result of previous stdize, from which scale and center values are taken. Column names are matched. This can be used for scaling new data using statistics of another data.
omit.cols	column names or numeric indices of columns that should be left unaltered.
prefix	either a logical value specifying whether the names of transformed columns should be prefixed, or a two-element character vector giving the prefixes. The prefixes default to "z." for scaled and "c." for centred variables.
append	logical, if TRUE, modified columns are appended to the original data frame.
response	logical, stating whether the response be standardized. By default only variables on the right-hand side of formula are standardized.
data	an object coercible to data.frame, containing the variables in formula. Passed to, and used by model.frame. For stdizeFit, a stdized data.frame to use.
...	for the formula method, additional arguments passed to model.frame. For other methods it is silently ignored.
object	a fitted model object or an expression being a call to the modelling function.
which	a character string naming arguments which should be modified. This should be all arguments which are evaluated in the data environment. Can be also TRUE to modify the expression as a whole. The data argument is additionally replaced with that passed to stdizeFit.
evaluate	if TRUE, the modified call is evaluated and the fitted model object is returned.
quote	if TRUE, avoids evaluating object. Equivalent to stdizeFit(quote(expr), ...). Defaults to NA in which case object being a call to non-primitive function is quoted.

Details

stdize resembles scale, but uses special rules for factors, similarly to standardize in package **arm**.

stdize differs from standardize in that it is used on data rather than on the fitted model object. The scaled data should afterwards be passed to the modelling function, instead of the original data.

Unlike standardize, it applies special 'binary' scaling only to two-level factors and logical variables, rather than to any variable with two unique values.

Variables of only one unique value are unchanged.

By default, stdize scales by dividing by standard deviation rather than twice the SD as standardize does. Scaling by SD is used also on uncentred values, which is different from scale where root-mean-square is used.

If center or scale are logical scalars or vectors of length equal to the number of columns of x, the centring is done by subtracting the mean (if center corresponding to the column is TRUE), and scaling is done by dividing the (centred) value by standard deviation (if corresponding scale is TRUE). If center or scale are numeric vectors with length equal to the number of columns of x (or numeric scalars for vector methods), then these are used instead. Any NAs in the numeric vector result in no centering or scaling on the corresponding column.

Note that scale = 0 is equivalent to no scaling (i.e. scale = 1).

Binary variables, logical or factors with two levels, are converted to numeric variables and transformed according to the argument binary, unless center or scale are explicitly given.

Value

`stdize` returns a vector or object of the same dimensions as `x`, where the values are centred and/or scaled. Transformation is carried out column-wise in `data.frames` and matrices.

The returned value is compatible with that of `scale` in that the numeric centring and scalings used are stored in attributes attributes `"scaled:center"` and `"scaled:scale"` (these can be `NA` if no centring or scaling has been done).

`stdizeFit` returns a modified, unevaluated call where the variable names are replaced to point the transformed variables, or if `evaluate` is `TRUE`, a fitted model object.

Author(s)

Kamil Bartoń

References

Gelman, A. (2008) Scaling regression inputs by dividing by two standard deviations. *Statistics in medicine* 27, 2865-2873.

See Also

Compare with `scale` and `standardize` or `rescale` (the latter two in package **arm**).

For typical standardizing, model coefficients transformation may be easier, see `std.coef`.

`apply` and `sweep` for arbitrary transformations of columns in a `data.frame`.

Examples

```
# compare "stdize" and "scale"
nmat <- matrix(runif(15, 0, 10), ncol = 3)

stdize(nmat)
scale(nmat)

rootmeansq <- function(v) {
    v <- v[!is.na(v)]
    sqrt(sum(v^2) / max(1, length(v) - 1L))
}

scale(nmat, center = FALSE)
stdize(nmat, center = FALSE, scale = rootmeansq)

if(require(lme4)) {
# define scale function as twice the SD to reproduce "arm::standardize"
twosd <- function(v) 2 * sd(v, na.rm = TRUE)

# standardize data (scaled variables are prefixed with "z.")
z.CO2 <- stdize(uptake ~ conc + Plant, data = CO2, omit = "Plant", scale = twosd)
summary(z.CO2)


fmz <- stdizeFit(lmer(uptake ~ conc + I(conc^2) + (1 | Plant)), data = z.CO2)
# produces:
# lmer(uptake ~ z.conc + I(z.conc^2) + (1 | Plant), data = z.CO2)
```

```
## standardize using scale and center from "z.CO2", keeping the original data:
z.CO2a <- stdize(CO2, source = z.CO2, append = TRUE)
# Here, the "subset" expression uses untransformed variable, so we modify only
# "formula" argument, keeping "subset" as-is. For that reason we needed the
# untransformed variables in "data".
stdizeFit(lmer(uptake ~ conc + I(conc^2) + (1 | Plant),
    subset = conc > 100,
    ), data = z.CO2a, which = "formula", evaluate = FALSE)


# create new data as a sequence along "conc"
newdata <-  data.frame(conc = seq(min(CO2$conc), max(CO2$conc), length = 10))

# scale new data using scale and center of the original scaled data:
z.newdata <- stdize(newdata, source = z.CO2)


# plot predictions against "conc" on real scale:
plot(newdata$conc, predict(fmz, z.newdata, re.form = NA))


# compare with "arm::standardize"
## Not run:
library(arm)
fms <- standardize(lmer(uptake ~ conc + I(conc^2) + (1 | Plant), data = CO2))
plot(newdata$conc, predict(fms, z.newdata, re.form = NA))

## End(Not run)
}
```

`subset.model.selection`

Subsetting model selection table

Description

Extract subset of a model selection table.

Usage

```
## S3 method for class 'model.selection'
subset(x, subset, select, recalc.weights = TRUE, recalc.delta = FALSE, ...)
## S3 method for class 'model.selection'
x[i, j, recalc.weights = TRUE, recalc.delta = FALSE, ...]
## S3 method for class 'model.selection'
x[[..., exact = TRUE]]
```

Arguments

`x`	a `model.selection` object to be subsetted.
`subset,select`	logical expressions indicating columns and rows to keep. See `subset`.

i,j	indices specifying elements to extract.
recalc.weights	logical value specyfying whether Akaike weights should be normalized across the new set of models to sum to one.
recalc.delta	logical value specyfying whether Δ_{IC} should be calculated for the new set of models (not done by default).
exact	logical, see [.
...	further arguments passed to [.data.frame (drop).

Details

Unlike the method for data.frame, single bracket extraction with only one index x[i] selects rows (models) rather than columns.

To select rows according to presence or absence of the variables (rather than their value), a pseudo-function has may be used with subset, e.g. subset(x, has(a, !b)) will select rows with *a* **and** without *b* (this is equivalent to !is.na(a) & is.na(b)). has can take any number of arguments.

Complex model terms need to be enclosed within curly brackets (e.g {s(a,k=2)}), except for within has. Backticks-quoting is also possible, but then the name must match exactly (including whitespace) the term name as returned by getAllTerms.

Enclosing in I prevents a name from being interpreted as column name.

To select rows where one variable can be present conditional on the presence of other variable(s), the function dc (**d**ependency **c**hain) can be used. dc takes any number of variables as arguments, and allows a variable to be included only if all the preceding arguments are also included (e.g. subset = dc(a, b, c) allows for models of form a, a+b and a+b+c but not b, c, b+c or a+c).

Value

A model.selection object containing only the selected models (rows). If columns are selected (via argument select or the second index x[, j]) and not all essential columns (i.e. all except "varying" and "extra") are present in the result, a plain data.frame is returned. Similarly, modifying values in the essential columns with [<-, [[<- or $<- produces a regular data frame.

Author(s)

Kamil Bartoń

See Also

dredge, subset and [.data.frame for subsetting and extracting from data.frames.

Examples

```
fm1 <- lm(formula = y ~ X1 + X2 + X3 + X4, data = Cement, na.action = na.fail)

# generate models where each variable is included only if the previous
# are included too, e.g. X2 only if X1 is there, and X3 only if X2 and X1
dredge(fm1, subset = dc(X1, X2, X3, X4))

# which is equivalent to
# dredge(fm1, subset = (!X2 | X1) & (!X3 | X2) & (!X4 | X3))

# alternatively, generate "all possible" combinations
ms0 <- dredge(fm1)
```

```
# ...and afterwards select the subset of models
subset(ms0, dc(X1, X2, X3, X4))
# which is equivalent to
# subset(ms0, (has(!X2) | has(X1)) & (has(!X3) | has(X2)) & (has(!X4) | has(X3)))

# Different ways of finding a confidence set of models:
# delta(AIC) cutoff
subset(ms0, delta <= 4, recalc.weights = FALSE)
# cumulative sum of Akaike weights
subset(ms0, cumsum(weight) <= .95, recalc.weights = FALSE)
# relative likelihood
subset(ms0, (weight / weight[1]) > (1/8), recalc.weights = FALSE)
```

`updateable` *Make a function return updateable result*

Description

Creates a function wrapper that stores a call in the object returned by its argument `FUN`.

Usage

```
updateable(FUN, eval.args = NULL, Class)

get_call(x)

## updateable wrapper for mgcv::gamm and gamm4::gamm4
uGamm(formula, random = NULL, ..., lme4 = inherits(random, "formula"))
```

Arguments

`FUN`	function to be modified, found via `match.fun`.
`eval.args`	optionally a character vector of function arguments' names to be evaluated in the stored call. See 'Details'.
`Class`	optional character vector naming class(es) to be set onto the result of `FUN` (not possible with formal S4 objects).
`x`	an object from which the call should be extracted.
`formula, random, ...`	arguments to be passed to `gamm` or `gamm4`
`lme4`	if `TRUE`, `gamm4` is called, `gamm` otherwise.

Details

Most model fitting functions in R return an object that can be updated or re-fitted via `update`. This is thanks to the `call` stored in the object, which can be used (possibly modified) later on. It is also utilised by `dredge` to generate sub-models. Some functions (such as `gamm` or `MCMCglmm`) do not provide their result with the `call` element. To work that around, `updateable` can be used on that function to store the call. The resulting "wrapper" should be used in exactly the same way as the original function.

Argument `eval.args` specifies names of function arguments that should be evaluated in the stored call. This is useful when, for example, the model object does not have `formula` element. The

default `formula` method tries to retrieve formula from the stored `call`, which works unless the formula has been given as a variable and value of that variable changed since the model was fitted (the last 'example' demonstrates this).

Value

`updateable` returns a function with the same arguments as `FUN`, wrapping a call to `FUN` and adding an element named `call` to its result if possible, otherwise an attribute `"call"` (if the returned value is atomic or a formal S4 object).

Note

`get_call` is similar to `getCall` (defined in package **stats**), but it can also extract the `call` when it is an `attribute` (and not an element of the object). Because the default `getCall` method cannot do that, the default `update` method will not work with atomic or S4 objects resulting from `updateable` wrappers.

`uGamm` sets also an appropriate class onto the result (`"gamm4"` and/or `"gamm"`), which is needed for some generics defined in **MuMIn** to work (note that unlike the functions created by `updateable` it has no formal arguments of the original function). As of version 1.9.2, `MuMIn::gamm` is no longer available.

Author(s)

Kamil Bartoń

See Also

`update`, `getCall`, `getElement`, `attributes`

`gamm`, `gamm4`

Examples

```
# Simple example with cor.test:

# From example(cor.test)
x <- c(44.4, 45.9, 41.9, 53.3, 44.7, 44.1, 50.7, 45.2, 60.1)
y <- c( 2.6,  3.1,  2.5,  5.0,  3.6,  4.0,  5.2,  2.8,  3.8)

ct1 <- cor.test(x, y, method = "kendall", alternative = "greater")

uCor.test <- updateable(cor.test)

ct2 <- uCor.test(x, y, method = "kendall", alternative = "greater")

getCall(ct1) # --> NULL
getCall(ct2)

#update(ct1, method = "pearson") --> Error
update(ct2, method = "pearson")
update(ct2, alternative = "two.sided")


## predefined wrapper for 'gamm':
```

```
set.seed(0)
dat <- gamSim(6, n = 100, scale = 5, dist = "normal")

fmm1 <- uGamm(y ~s(x0)+ s(x3) + s(x2), family = gaussian, data = dat,
    random = list(fac = ~1))

getCall(fmm1)
class(fmm1)

###

## Not run:
library(caper)
data(shorebird)
shorebird <- comparative.data(shorebird.tree, shorebird.data, Species)

fm1 <- crunch(Egg.Mass ~ F.Mass * M.Mass, data = shorebird)

uCrunch <- updateable(crunch)

fm2 <- uCrunch(Egg.Mass ~ F.Mass * M.Mass, data = shorebird)

getCall(fm1)
getCall(fm2)
update(fm2) # Error with 'fm1'
dredge(fm2)

## End(Not run)

###
## Not run:
# "lmekin" does not store "formula" element
library(coxme)
uLmekin <- updateable(lmekin, eval.args = "formula")

f <- effort ~ Type + (1|Subject)
fm1 <- lmekin(f, data = ergoStool)
fm2 <- uLmekin(f, data = ergoStool)

f <- wrong ~ formula # reassigning "f"

getCall(fm1) # formula is "f"
getCall(fm2)

formula(fm1) # returns the current value of "f"
formula(fm2)

## End(Not run)
```

`Weights` *Akaike weights*

Description

Calculate, extract or set normalized model likelihoods ('Akaike weights').

Usage

```
Weights(x)
Weights(x) <- value
```

Arguments

x	a numeric vector of information criterion values such as AIC, or objects returned by functions like AIC. There are also methods for extracting 'Akaike weights' from "model.selection" or "averaging" objects.
value	numeric, the new weights for the "averaging" object or NULL to reset the weights based on the original IC used.

Details

The replacement function can assign new weights to an "averaging" object, affecting coefficient values and order of component models.

Value

For the extractor, a numeric vector of normalized likelihoods.

Note

On assigning new weights, the model order changes accordingly, so assigning the same weights again will cause incorrect re-calculation of averaged coefficients. To avoid that, either re-set model weights by assigning NULL, or use ordered weights.

Author(s)

Kamil Bartoń

See Also

importance, weighted.mean

armWeights, bootWeights, BGWeights, cos2Weights, jackknifeWeights and stackingWeights can be used to produce model weights.

weights, which extracts fitting weights from model objects.

Examples

```
fm1 <- glm(Prop ~ dose, data = Beetle, family = binomial)
fm2 <- update(fm1, . ~ . + I(dose^2))
fm3 <- update(fm1, . ~ log(dose))
fm4 <- update(fm3, . ~ . + I(log(dose)^2))

round(Weights(AICc(fm1, fm2, fm3, fm4)), 3)


am <- model.avg(fm1, fm2, fm3, fm4, rank = AICc)

coef(am)

# Assign equal weights to all models:
```

```
Weights(am) <- rep(1, 4) # assigned weights are rescaled to sum to 1
Weights(am)
coef(am)

# Assign dummy weights:
wts <- c(2,1,4,3)
Weights(am) <- wts
coef(am)
# Component models are now sorted according to the new weights.
# The same weights assigned again produce incorrect results!
Weights(am) <- wts
coef(am) # wrong!
#
Weights(am) <- NULL # reset to original model weights
Weights(am) <- wts
coef(am) # correct
```

Index