Package 'marked'

March 30, 2018

Version 1.2.1

Date 2018-03-29

Title Mark-Recapture Analysis for Survival and Abundance Estimation

Author Jeff Laake <jefflaake@gmail.com>, Devin Johnson

<devin.johnson@noaa.gov>, Paul Conn <paul.conn@noaa.gov>, example for simHMM from Jay Rotella

Maintainer Jeff Laake <jefflaake@gmail.com>

Description Functions for fitting various models to capture-recapture data including mixed-effects Cormack-Jolly-Seber(CJS) and multistate models and the multi-variate state model structure for survival estimation and POPAN structured Jolly-Seber models for abundance estimation. There are also Hidden Markov model (HMM) implementations of CJS and multistate models with and without state uncertainty and a simulation capability for HMM models.

Depends R (>= 3.2.0), lme4, methods, parallel

Imports graphics, stats, utils, R2admb, truncnorm, coda, Matrix, numDeriv, expm, Rcpp (>= 0.9.13), TMB, optimx (>= 2013.8.6)

Suggests ggplot2

LinkingTo Rcpp

SystemRequirements ADMB version 11 <http://admb-project.org/> for use.admb=TRUE; see readme.txt

LazyLoad yes

License GPL (>= 2)

RoxygenNote 6.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-03-30 12:52:22 UTC

R topics documented:

`backward_prob` *Computes backward probabilities*

Description

Computes backward probability sequence for a set of capture histories

Usage

```
backward_prob(object, ddl = NULL)
```

Arguments

`object`	fitted crm model (must be an HMM model)
`ddl`	design data list

Value

array of backward probabilities (one for each id, state, occasion)

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. See page 61.

Examples

```
#

# This example is excluded from testing to reduce package check time
# cormack-jolly-seber model
data(dipper)
mod=crm(dipper,model="hmmcjs")
backward_prob(mod)
```

---

cjs.accumulate *Accumulates common capture history values*

Description

To speed up compuation, animals with the same capture history and design matrix are accumulated and represented by a frequency. Computes starting values for Phi and p parameters from the list of design matrices and the summarized data list including ch matrix and first and last vectors. If any values are missing (NA) or abs(par)>5, they are set to 0.

Usage

```
cjs.accumulate(x, model_data, nocc, freq, chunk_size)
```

Arguments

x	data
model_data	list of design matrices, fixed parameters and time intervals all which can vary by animal
nocc	number of capture occasions
freq	frequency of each capture history before accumulation
chunk_size	size that determines number of pieces of data/design matrix that are handled. Smaller chunk_size requires more time but less memory. 1e7 is default set in cjs.

Value

modified model_data list that is accumulated

Author(s)

Jeff Laake

cjs.hessian *Compute variance-covariance matrix for fitted CJS model*

Description

A wrapper function that sets up call to hessian function to compute and then invert the hessian.

Usage

```
cjs.hessian(model)
```

Arguments

model	fitted CJS model from function crm

Value

variance-covariance matrix for specified model or the model object with the stored vcv depending on whether the model has already been run

Author(s)

Jeff Laake

cjs.initial *Computes starting values for CJS p and Phi parameters*

Description

Computes starting values for Phi and p parameters from the list of design matrices and the summarized data list including ch matrix and first and last vectors. If any values are missing (NA) or abs(par)>5, they are set to 0.

Usage

```
cjs.initial(dml, imat, link = "logit")
```

Arguments

dml	design matrix list for Phi and p
imat	list containing chmat, first and last
link	either "logit" (for cjs) or "probit" (for probitCJS)

Value

list of initial parameter estimates

Author(s)

Jeff Laake

`cjs.lnl` *Likelihood function for Cormack-Jolly-Seber model*

Description

For a given set of parameters and data, it computes -log Likelihood value.

Usage

```
cjs.lnl(par, model_data, Phi.links = NULL, p.links = NULL, debug = FALSE,
  all = FALSE, cjsenv)
```

Arguments

`par`	vector of parameter values
`model_data`	a list that contains: 1)imat-list of vectors and matrices constructed by `process.ch` from the capture history data, 2)Phi.dm design matrix for Phi constructed by `create.dm`, 3)p.dm design matrix for p constructed by `create.dm`, 4)Phi.fixed matrix with 3 columns: ch number(i), occasion number(j), fixed value(f) to fix phi(i,j)=f, 5)p.fixed matrix with 3 columns: ch number(i), occasion number(j), and 6) time.intervals intervals of time between occasions if not all 1 fixed value(f) to fix p(i,j)=f
`Phi.links`	vector of links for each parameter
`p.links`	vector of links for each parameter
`debug`	if TRUE will printout values of par and function value
`all`	if TRUE, returns entire list rather than just lnl; can be used to extract reals
`cjsenv`	environment for cjs to update iteration counter

Details

This function uses a FORTRAN subroutine (cjs.f) to speed up computation of the likelihood but the result can also be obtained wholly in R with a small loss in precision. See R code below. The R and FORTRAN code uses the likelihood formulation of Pledger et al.(2003).

```
get.p=function(beta,dm,nocc,Fplus)
{
# compute p matrix from parameters (beta) and list of design matrices (dm)
# created by function create.dm
  ps=cbind(rep(1,nrow(dm)/(nocc-1)),
     matrix(dm
  ps[Fplus==1]=plogis(ps[Fplus==1])
  return(ps)
```

```
}
get.Phi=function(beta,dm,nocc,Fplus)
{
# compute Phi matrix from parameters (beta) and list of design matrices (dm)
# created by function create.dm
  Phis=cbind(rep(1,nrow(dm)/(nocc-1)),
     matrix(dm
  Phis[Fplus==1]=plogis(Phis[Fplus==1])
  return(Phis)
}
##############################################################################
# cjs.lnl - computes likelihood for CJS using Pledger et al (2003)
# formulation for the likelihood. This code does not cope with fixed parameters or
# loss on capture but could be modified to do so. Also, to work directly with cjs.r and
# cjs.accumulate call to process.ch would have to have all=TRUE to get Fplus and L.
# Arguments:
# par             - vector of beta parameters
# imat         - list of freq, indicator vector and matrices for ch data created by process.ch
# Phi.dm         - list of design matrices; a dm for each capture history
# p.dm           - list of design matrices; a dm for each capture history
# debug          - if TRUE show iterations with par and -2lnl
# time.intervals - intervals of time between occasions
# Value: -LnL using
##############################################################################
cjs.lnl=function(par,model_data,Phi.links=NULL,p.links=NULL,debug=FALSE,all=FALSE) {
if(debug)cat("\npar = ",par)
#extract Phi and p parameters from par vector
nphi=ncol(model_data$Phi.dm)
np=ncol(model_data$p.dm)
beta.phi=par[1:nphi]
beta.p=par[(nphi+1):(nphi+np)]
#construct parameter matrices (1 row for each capture history and a column
 #for each occasion)
Phis=get.Phi(beta.phi,model_data$Phi.dm,nocc=ncol(model_data$imat$chmat),
                          model_data$imat$Fplus)
if(!is.null(model_data$time.intervals))
{
exponent=cbind(rep(1,nrow(Phis)),model_data$time.intervals)
Phis=Phis^exponent
}
ps=get.p(beta.p,model_data$p.dm,nocc=ncol(model_data$imat$chmat),
           model_data$imat$Fplus)
if(debug)cat("\npar = ",par)
# Compute probability of dying in interval from Phis
M=cbind((1-Phis)[,-1],rep(1,nrow(Phis)))
# compute cummulative survival from release across each subsequent time
# and the cummulative probability for detection (capture) across each time
Phi.cumprod=1-model_data$imat$Fplus + Phis*model_data$imat$Fplus
```

```
cump=(1-model_data$imat$Fplus)+model_data$imat$Fplus*
         (model_data$imat$chmat*ps+(1-model_data$imat$chmat)*(1-ps))
for (i in 2:ncol(cump))
{
Phi.cumprod[,i]=Phi.cumprod[,i-1]*Phi.cumprod[,i]
cump[,i]=cump[,i-1]*cump[,i]
}
# compute prob of capture-history
pch=rowSums(model_data$imat$L*M*Phi.cumprod*cump)
lnl=-sum(model_data$imat$freq*log(pch))
if(debug)cat("\n-2lnl = ",2*lnl)
return(lnl)
}
```

Value

either -log likelihood value if `all=FALSE` or the entire list contents of the call to the FORTRAN subroutine if `all=TRUE`. The latter is used from `cjs_admb` after optimization to extract the real parameter estimates at the final beta values.

Author(s)

Jeff Laake

References

Pledger, S., K. H. Pollock, et al. (2003). Open capture-recapture models with heterogeneity: I. Cormack-Jolly-Seber model. Biometrics 59(4):786-794.

`cjs_admb` *Fitting function for CJS models*

Description

A function for computing MLEs for a specified Cormack-Jolly-Seber open population capture-recapture model for processed dataframe `x` with user specified formulas in `parameters` that create list of design matrices `dml`. This function can be called directly but is most easily called from `crm` that sets up needed arguments.

Usage

```
cjs_admb(x, ddl, dml, model_data = NULL, parameters, accumulate = TRUE,
  initial = NULL, method, hessian = FALSE, debug = FALSE,
  chunk_size = 1e+07, refit, itnmax = NULL, control = NULL, scale,
  use.admb = FALSE, crossed = TRUE, compile = FALSE, extra.args = NULL,
  reml, clean = TRUE, ...)
```

Arguments

x	processed dataframe created by process.data
ddl	list of dataframes for design data; created by call to make.design.data
dml	list of design matrices created by create.dm from formula and design data
model_data	a list of all the relevant data for fitting the model including imat, Phi.dm,p.dm,Phi.fixed,p.fixed, and time.intervals. It is used to save values and avoid accumulation again if the model was re-rerun with an additional call to cjs when using autoscale or re-starting with initial values. It is stored with returned model object.
parameters	equivalent to model.parameters in crm
accumulate	if TRUE will accumulate capture histories with common value and with a common design matrix for Phi and p to speed up execution
initial	list of initial values for parameters if desired; if each is a named vector from previous run it will match to columns with same name
method	method to use for optimization; see optim
hessian	if TRUE will compute and return the hessian
debug	if TRUE will print out information for each iteration
chunk_size	specifies amount of memory to use in accumulating capture histories; amount used is 8*chunk_size/1e6 MB (default 80MB)
refit	non-zero entry to refit
itnmax	maximum number of iterations
control	control string for optimization functions
scale	vector of scale values for parameters
use.admb	if TRUE creates data file for admbcjs.tpl and runs admb optimizer
crossed	if TRUE it uses cjs.tpl or cjs_reml.tpl if reml=FALSE or TRUE respectively; if FALSE, then it uses cjsre which can use Gauss-Hermite integration
compile	if TRUE forces re-compilation of tpl file
extra.args	optional character string that is passed to admb if use.admb==TRUE
reml	if set to TRUE uses cjs_reml if crossed
clean	if TRUE, deletes the tpl and executable files for amdb if use.admb=T
...	any remaining arguments are passed to additional parameters passed to optim or cjs.lnl

Details

It is easiest to call cjs through the function crm. Details are explained there.

Be cautious with this function at present. It does not include many checks to make sure values like fixed values will remain in the specified range of the data. Normally this would not be a big problem but because cjs.lnl calls an external FORTRAN subroutine, if it gets a subscript out of bounds, it will cause R to terminate. So make sure to save your workspace frequently if you use this function in its current implementation.

Value

The resulting value of the function is a list with the class of crm,cjs such that the generic functions print and coef can be used. Elements are 1) beta: named vector of parameter estimatesm 2) lnl: -2*log likelihood, 3) AIC: lnl + 2* number of parameters, 4) convergence: result from `optim`; if 0 `optim` thinks it converged, 5) count:`optim` results of number of function evaluations, 6) reals: dataframe of data and real Phi and p estimates for each animal-occasion excluding those that occurred before release, 7) vcv:var-cov matrix of betas if hessian=TRUE was set.

Author(s)

Jeff Laake

References

Pledger, S., K. H. Pollock, et al. (2003). Open capture-recapture models with heterogeneity: I. Cormack-Jolly-Seber model. Biometrics 59(4):786-794.

`cjs_delta` *HMM Initial state distribution functions*

Description

Functions that compute the initial probability distribution for the states. Currently only CJS, MS models and MS models with state uncertainty are included and these all use cjs_delta to assign a known state.

Usage

```
cjs_delta(pars, m, F, T, start)
```

Arguments

`pars`	list of real parameter matrices (id by occasion) for each type of parameter
`m`	number of states
`F`	initial occasion vector
`T`	number of occasions
`start`	matrix with values that are first occasion and for some CJS type models the state of first observation

Value

2-d array of initial state probability vectors for each id

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. 275p.

`cjs_gamma` *HMM Transition matrix functions*

Description

Functions that compute the transition matrix for various models. Currently only CJS and MS models are included.

Usage

```
cjs_gamma(pars, m, F, T)
```

Arguments

`pars`	list of real parameter values for each type of parameter
`m`	number of states
`F`	initial occasion vector
`T`	number of occasions

Value

array of id and occasion-specific transition matrices - Gamma in Zucchini and MacDonald (2009)

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. 275p.

cjs_tmb *Fitting function for CJS models*

Description

A function for computing MLEs for a specified Cormack-Jolly-Seber open population capture-recapture model for processed dataframe x with user specified formulas in parameters that create list of design matrices dml. This function can be called directly but is most easily called from crm that sets up needed arguments.

Usage

```
cjs_tmb(x, ddl, dml, model_data = NULL, parameters, accumulate = TRUE,
  initial = NULL, method, hessian = FALSE, debug = FALSE,
  chunk_size = 1e+07, refit, itnmax = NULL, control = NULL, scale,
  crossed = TRUE, compile = TRUE, extra.args = NULL, reml,
  clean = FALSE, getreals = FALSE, prior = FALSE, prior.list = NULL,
  tmbfct = "f1", ...)
```

Arguments

x	processed dataframe created by process.data
ddl	list of dataframes for design data; created by call to make.design.data
dml	list of design matrices created by create.dm from formula and design data
model_data	a list of all the relevant data for fitting the model including imat, Phi.dm,p.dm,Phi.fixed,p.fixed, and time.intervals. It is used to save values and avoid accumulation again if the model was re-rerun with an additional call to cjs when using autoscale or re-starting with initial values. It is stored with returned model object.
parameters	equivalent to model.parameters in crm
accumulate	if TRUE will accumulate capture histories with common value and with a common design matrix for Phi and p to speed up execution
initial	list of initial values for parameters if desired; if each is a named vector from previous run it will match to columns with same name
method	method to use for optimization; see optim
hessian	if TRUE will compute and return the hessian
debug	if TRUE will print out information for each iteration
chunk_size	specifies amount of memory to use in accumulating capture histories; amount used is 8*chunk_size/1e6 MB (default 80MB)
refit	non-zero entry to refit
itnmax	maximum number of iterations
control	control string for optimization functions
scale	vector of scale values for parameters

crossed	if TRUE it uses cjs.tpl or cjs_reml.tpl if reml=FALSE or TRUE respectively; if FALSE, then it uses cjsre which can use Gauss-Hermite integration
compile	if TRUE forces re-compilation of tpl file
extra.args	optional character string that is passed to admb if use.admb==TRUE
reml	if set to TRUE uses cjs_reml if crossed
clean	if TRUE, deletes the tpl and executable files for amdb if use.admb=T
getreals	if TRUE, will compute real Phi and p values and std errors
prior	if TRUE will expect vectors of prior values in list prior.list
prior.list	which contains for normal distributions 1) mu_phi_prior: vector of mu values for phi_beta, 2) sigma_phi_prior: vector of sigma values for phi_beta, 3) mu_p_prior: vector of mu values for p_beta, 4) sigma_p_prior: vector of sigma values for p_beta, 5) random_mu_phi_prior: vector of mu values for ln sigma of random effects, 6) random_sigma_phi_prior: vector of sigma values for ln sigma_phi, 7) random_mu_p_prior: vector of mu values for ln sigma_p, 8) random_sigma_p_prior: vector of sigma values for ln sigma_p.
tmbfct	either "f1" - default or "f2" - any random effects treated as fixed effects or "f3" fixed effects fixed at mode and no random effects.
...	any remaining arguments are passed to additional parameters passed to optim or cjs.lnl

Details

It is easiest to call cjs through the function crm. Details are explained there.

Be cautious with this function at present. It does not include many checks to make sure values like fixed values will remain in the specified range of the data. Normally this would not be a big problem but because cjs.lnl calls an external FORTRAN subroutine, if it gets a subscript out of bounds, it will cause R to terminate. So make sure to save your workspace frequently if you use this function in its current implementation.

Value

The resulting value of the function is a list with the class of crm,cjs such that the generic functions print and coef can be used. Elements are 1) beta: named vector of parameter estimatesm 2) lnl: -2*log likelihood, 3) AIC: lnl + 2* number of parameters, 4) convergence: result from optim; if 0 optim thinks it converged, 5) count:optim results of number of function evaluations, 6) reals: dataframe of data and real Phi and p estimates for each animal-occasion excluding those that occurred before release, 7) vcv:var-cov matrix of betas if hessian=TRUE was set.

Author(s)

Jeff Laake

References

Pledger, S., K. H. Pollock, et al. (2003). Open capture-recapture models with heterogeneity: I. Cormack-Jolly-Seber model. Biometrics 59(4):786-794.

coef.crm *Extract coefficients*

Description

Extracts the beta coefficients from the model results.

Usage

```
## S3 method for class 'crm'
coef(object,...)
```

Arguments

object	crm model result
...	generic arguments not used here

Value

returns a dataframe with estimates and standard errors and confidence intervals if hessian=TRUE on model run.

Author(s)

Jeff Laake

See Also

crm

compute.real *Compute estimates of real parameters*

Description

Computes real estimates and their var-cov for a particular parameter.

Usage

```
compute.real(model, parameter, ddl = NULL, dml = NULL, unique = TRUE,
  vcv = FALSE, se = FALSE, chat = 1, subset = NULL, select = NULL,
  showDesign = FALSE, include = NULL, uselink = FALSE)
```

Arguments

model	model object
parameter	name of real parameter to be computed (eg "Phi" or "p")
ddl	list of design data
dml	design matrix list
unique	TRUE if only unique values should be returned
vcv	logical; if TRUE, computes and returns v-c matrix of real estimates
se	logical; if TRUE, computes std errors and conf itervals of real estimates
chat	over-dispersion value
subset	logical expression using fields in real dataframe
select	character vector of field names in real that you want to include
showDesign	if TRUE, show design matrix instead of data
include	vector of field names always to be included even when select or unique specified
uselink	default FALSE; if TRUE uses link values in evaluating uniqueness

Details

This code is complicated because it handles both the MCMC models and the likelihood models. The former is quite simple than the latter because all of the real computation is done by the model code and this function only computes summaries. The likelihood model code is complicated primarily by the mlogit parameters which are computed in 2 stages: 1) log link and 2) summation to normalize. The mlogit is handled differently depending on the model. For MS and JS models, one of the parameters is computed by subtraction (specified as addone==TRUE) whereas the HMM models (addone=FALSE) specify a parameter for each cell and one is fixed by the user to 1. The latter is preferable because it then provides an estimate and a std error for each parameter whereas the subtracted value is not provided for MS and JS.

This function differs from compute.real in RMark because it only computes the values for a single parameter whereas the function with the same name in RMark can compute estimates from multiple parameters (eg Phi and p).

Value

A data frame (`real`) is returned if `vcv=FALSE`; otherwise, a list is returned also containing vcv.real:

real	data frame containing estimates, and if vcv=TRUE it also contains standard errors and confidence intervals
vcv.real	variance-covariance matrix of real estimates

Author(s)

Jeff Laake

Examples

```
data(dipper)
dipper.proc=process.data(dipper,model="cjs",begin.time=1)
dipper.ddl=make.design.data(dipper.proc)
mod.Phisex.pdot=crm(dipper.proc,dipper.ddl,
 model.parameters=list(Phi=list(formula=~sex+time),p=list(formula=~1)),hessian=TRUE)
xx=compute.real(mod.Phisex.pdot,"Phi",unique=TRUE,vcv=TRUE)
```

---

compute_matrices *Compute HMM matrices*

Description

Computes gamma, dmat and delta (initial) matrices(arrays) and returns them in a list.

Usage

```
compute_matrices(object, ddl = NULL)
```

Arguments

object	fitted crm model (must be an HMM model)
ddl	design data list

Value

list with gamma, dmat and delta arrays

Author(s)

Jeff Laake

---

convert.link.to.real *Convert link values to real parameters*

Description

Computes real parameters from link values

Usage

```
convert.link.to.real(x, model = NULL, links = NULL, fixed = NULL)
```

Arguments

x	Link values to be converted to real parameters
model	model object
links	vector of character strings specifying links to use in computation of reals
fixed	vector of fixed values for real parameters that are needed for calculation of reals from mlogits when some are fixed

Details

Computation of the real parameter from the link value is relatively straightforward for most links and the function inverse.link is used. The only exception is parameters that use the mlogit link which requires the transformation across sets of parameters. This is a convenience function that does the necessary work to convert from link to real for any set of parameters. The appropriate links are obtained from model$links unless the argument links is specified and they will over-ride those in model.

Value

vector of real parameter values

Author(s)

Jeff Laake

See Also

inverse.link,compute.real

---

create.dm *Creates a design matrix for a parameter*

Description

Creates a design matrix using the design dataframe, a formula and any intervals defined for time, cohort and age.

Usage

```
create.dm(x, formula, time.bins=NULL, cohort.bins=NULL, age.bins=NULL,
              chunk_size=1e7, remove.intercept=NULL,remove.unused.columns=TRUE)

      create.dml(ddl,model.parameters,design.parameters,restrict=FALSE,
         chunk_size=1e7,use.admb=FALSE,remove.unused.columns=TRUE,simplify=FALSE)
```

Arguments

x	design dataframe created by create.dmdf
formula	formula for model in R format
time.bins	any bins of time to collapse values
cohort.bins	any bins of cohort to collapse values
age.bins	any bins of cohort to collapse values
chunk_size	specifies amount of memory to use in creating design matrices; amount used is 8*chunk_size/1e6 MB (default 80MB)
remove.intercept	if TRUE, forces removal of intercept in design matrix
remove.unused.columns	if TRUE, unused columns are removed; otherwise they are left
ddl	Design data list which contains a list element for each parameter type; if NULL it is created
design.parameters	Specification of any grouping variables for design data for each parameter
model.parameters	List of model parameter specifications
restrict	if TRUE, only use design data with Time >= Cohort
use.admb	if TRUE uses mixed.model.admb for random effects; otherwise mixed.model
simplify	if TRUE simplies real parameter structure for some models; at this time it is not more efficient so ignore

Value

create.dm returns a fixed effect design matrix constructed with the design dataframe and the formula for a single parametre. It excludes any columns that are all 0. create.dml returns a list with an element for for each parameter with a sub-list for the fixed effect (fe) and random effects. The re structure depends on switch use.admb. When TRUE, it contains a single design matrix (re.dm) and indices for random effects (re.indices). When FALSE, it returns re.list which is a list with an element for each random component containing re.dm and indices for that random effect (eg (1|id) + (1|time) would produce elements for id and time.

Author(s)

Jeff Laake

`create.dmdf` *Creates a dataframe with all the design data for a particular parameter in a crm model*

Description

Creates a dataframe with all the design data for a particular parameter in a crm model which currently includes "cjs" or "js". These design data are fundamentally different than the design data created for `mark` models as explained below.

Usage

```
create.dmdf(x, parameter, time.varying = NULL, fields = NULL)
```

Arguments

`x`	processed dataframe from function `process.data`
`parameter`	list with fields defining each values for each parameter; as created by `setup.parameters`
`time.varying`	vector of field names that are time-varying for this parameter
`fields`	character vector containing field names for variables in x to be included in design matrix dataframe; if NULL all other than ch are included

Details

This function is intended to be called from `make.design.data`. It takes the data in `x` and creates a dataframe with all of the data needed to fit capture-recapture models (crm) which currently includes "cjs" (Cormack-Jolly-Seber) or "js" (POPAN formulation of the Jolly-Seber model). Before jumping into the details it is useful to have an understanding of the differences between MARK (via the `mark in RMark` function) and the package `mra` written by Trent McDonald and how they relate to the implementation in `cjs_admb`. With MARK, animals can be placed in groups and the parameters for the model specified via PIMs (parameter index matrices) link the parameters to the specific animals. For example, if for a particular group the `Phi` PIM is

```
1 2 3
  4 5
    6
```

Then animals in that group that were first caught/released on occasion 1 have the parameters 1,2,3 for the 3 occasions. Those first caught/released on occasion 2 have parameters 4 and 5 for occasions 2 and 3 and those first caught/released on occasion 3 have parameter 6 for occasion 3. Another group of animals would have a different set of indices for the same set of parameters so they could be discriminated. Many people find this rather confusing until they get used to it but even then if you have many different groups and many occasions, then the indexing process is prone to error. Thus, the rationale for RMark which automates the PIM construction and its use is largely transparent to the user. What RMark does is to create a data structure called design data that automatically assigns `design data` to the indices in the PIM. For example, 1 to 6 would be given the data used to create

that group and 1 to 3 would be assigned to cohort 1, ... and 1 would be assigned to occasion 1 and 2 and 4 would be assigned to occasion 2 etc. It also creates an age field which follows the diagonals and can be initialized with the intial age at the time of first capture which is group specific. With a formula and these design data a design matrix can be constructed for the model where the row in the design matrix is the parameter index (e.g., the first 6 rows would be for parameters 1 to 6 as shown above). That would be all good except for individual covariates which are not group-specific. MARK handles individual covariates by specifying the covariate name (eg "weight") as a string in the design matrix. Then for each capture history in plugs in the actual covariate values for that animal to complete the design matrix for that animal. For more details see Laake and Rexstad (2008).

From a brief look at package `mra` and personal communication with Trent McDonald, I give the following brief and possibly incorrect description of the pacakge `mra` at the time of writing (28 Aug 2008). In that package, the whole concept of PIMS is abandoned and instead covariates are constructed for each occasion for each animal. Thus, each animal is effectively put in its own group and it has a parameter for each occasion. This novel approach is quite effective at blurring the lines between design data and individual covariate data and it removes the needs for PIMS because each animal (or unique capture history) has a real parameter for each occasion. The downside of the pacakge `mra` is that every covariate is assumed to be time-varying and any factor variables like `time` are coded manually as dummy variables for each level rather than using the R facilities for handling factor variables in the formula to create the design matrix.

In the `crm,cjs_admb,js` functions in this package I have used basic idea in `mra` but I have taken a different approach to model development that allows for time-varying covariates but does not restrict each covariate to be time-varying and factor variables are used as such which removes the need to construct dummy variables; although the latter could still be used. First an example is in order to explain how this all works. Consider the follwing set of capture histories for small capture-recapture data set with 4 capture occasions:

```
1001 0111 0011
```

To relate the current structure to the concept of PIMS I define the following matrix

```
1 2 3
 4 5 6
 7 8 9
```

If you think of these as `Phi` parameter indices, then 1 to 3 are survivals for the intervals 1-2,2-3,3-4 for animal 1, and 4-6 and 7-9 are the same for animals 2 and 3. This matrix would have a row for each animal. Now you'll notice that for animal 2 parameter 4 is not needed and for animal 3, parameters 7 and 8 are not needed because they are prior to their entry in the study. While that is certainly true there is no harm in having them and the advantage comes in being able to have a complete matrix in R rather than having a triangular one.

So now we are finally getting to the description of what this function does. It constructs a dataframe with a row for each animal-occasion. Following on with the example above, depending on how the arguments are set the following dataframe could be constructed:

```
row time Time cohort Cohort age Age initial.age
```

```
1    1    0    1    0    0    0    0
2    2    1    1    0    1    1    0
3    3    2    1    0    2    2    0
4    1    0    2    1    0    0    0
5    2    1    2    1    1    1    0
6    3    2    2    1    2    2    0
7    1    0    3    2    0    0    0
8    2    1    3    2    1    1    0
9    3    2    3    2    2    2    0
```

The fields starting with a lowercase character (time,cohort,age) are created as factor variables and those with an uppercase are created as numeric variables. Note: the age field is bounded below by the minimum `initial.age` to avoid creating factor levels with non-existent data for occasions prior to first capture that are not used. For example, an animal first caught on occasion 2 with an `initial.age=0` is technically -1 on occasion 1 with a `time.interval` of 1. However, that parameter would never be used in the model and we don't want a factor level of -1.

A formula of ~time would create a design matrix with 3 columns (one for each factor level) and ~Time would create one with 2 columns with the first being an intercept and the second with the numeric value of Time.

Now here is the simplicity of it. The following few expressions in R will convert this dataframe into a matrix of real parameters (assuming `beta=c(1,1,1)` that are structured like the square PIM matrix without the use of PIMs.

```
nocc=4
x=data.frame(ch=c("1001","0111","0011"),stringsAsFactors=FALSE)
beta=c(1,1,1)
x.proc=process.data(x,model="cjs")
Phi.dmdf=make.design.data(x.proc)$Phi
Phi.dm=create.dm(Phi.dmdf,~time)
Phimat=matrix(plogis(Phi.dm
```

Note that the order of the columns for `Phi.dmdf` differs slightly from what is shown above. Also, `plogis` is an R function that computes the inverse-logit. Once you have the matrix of `Phi` and `p` values the calculation of the likelihood is straightforward using the formulation of Pledger et al. (2003) (see `cjs.lnl`). The values in the design dataframe are not limited to these fields. The 2 arguments `time.varying` and `fields` are vectors of character strings which specify the names of the dataframe columns in `x` that should be included. For example if `x` contained a field `sex` with the values "M","F","M" for the 3 records in our example, and the argument `fields=c("sex")` was used then a column named `sex` would be included in design dataframe with the values "M","M","M","F","F","F","M","M","M". The value of the column `sex` in `x` is repeated for each of the occasions for that animal(capture history). Now if the value of the field changes for each occasion then we use the argument `time.varying` instead. To differentiate the values in the dataframe `x` the columns are named with an occasion number. For example, if the variable was named `cov` and it was to be used for `Phi`, then the variables would be named `cov1,cov2,cov3` in `x`. Let's say that x was structured as follows:

```
ch   cov1 cov2 cov3
1001   1   0     1
0111   0   2     1
0011   0   0     0
```

If you specified the argument `time.varying=c("cov")` then in the design dataframe a field named `cov` would be created and the values would be `1,0,1,0,2,1,0,0,0`. Thus the value is both animal and occasion specific. Had the covariate been used for `p` then they would be named `cov2,cov3,cov4` because the covariate is for those occasions for `p` whereas for `Phi` the covariate is labelled with the occasion that begins the interval. Any number of fields can be specified in `fields` and `time.varying` that are specified in `x`.

The input dataframe `x` has a few minor requirements on its structure. First, it must contain a field called `ch` which contains the capture-history as a string. Note that in general strings are converted to factor variables by default when they are put into a dataframe but as shown above that can be controlled by the argument `stringsAsFactors=FALSE`. The capture history should be composed only of 0 or 1 and they should all be the same length (at present no error checking on this). Although it is not necessary, the dataframe can contain a field named `freq` which specifies the frequency of that capture history. If the value of `freq` is negative then these are treated as loss on capture at the final capture occasion (last 1). If `freq` is missing then a value of 1 is assumed for all records. Another optional field is `initial.age` which specifies the age of the animal at the time it was first captured. This field is used to construct the `age` and `Age` fields in the design dataframe. The default is to assume `initial.age=0` which means the `age` is really time since first marked. Any other fields in `x` are user-specified and can be a combination of factor and numeric variables that are either time-invariant or time-varying (and named appropriately).

The behavior of create.dmdf can vary depending on the values of begin.time and time.intervals. An explanation of these values and how they can be used is given in `process.data`.

Value

A dataframe with all of the individual covariate data and the standard design data of time, Time, cohort, Cohort, age and Age; where lowercase first letter implies a factor and uppercase is a numeric variable for those variables.

Author(s)

Jeff Laake

References

Laake, J. and E. Rexstad (2007). RMark – an alternative approach to building linear models in MARK. Program MARK: A Gentle Introduction. E. Cooch and G. C. White.

Pledger, S., K. H. Pollock, et al. (2003). Open capture-recapture models with heterogeneity: I. Cormack-Jolly-Seber model. Biometrics 59(4):786-794.

`create.fixed.matrix` *Create parameters with fixed matrix*

Description

Creates fixed matrix in parameters from ddl$fix values

Usage

```
create.fixed.matrix(ddl, parameters)
```

Arguments

ddl	design data list
parameters	parameter specifiifcation list

Value

parameters with fixed matrix set

Author(s)

Jeff Laake

---

create.links *Creates a 0/1 vector for real parameters with sin link*

Description

For each row in a given design matrix it assigns a value 1, if the columns used in the design matrix are only used as an identity matrix (only one 1 and remaining columns all 0.

Usage

```
create.links(dm)
```

Arguments

dm	design matrix

Value

A vector of length=nrow(dm) with value of 0 except for rows that can accommodate a sin link (identity design matrix). This function is not currently used because it has not been thoroughly tested.

Author(s)

Jeff Laake

crm *Capture-recapture model fitting function*

Description

Fits user specified models to some types of capture-recapture wholly in R and not with MARK. A single function that processes data, creates the design data, makes the crm model and runs it

Usage

```
crm(data, ddl = NULL, begin.time = 1, model = "CJS", title = "",
  model.parameters = list(), design.parameters = list(), initial = NULL,
  groups = NULL, time.intervals = NULL, debug = FALSE, method = "BFGS",
  hessian = FALSE, accumulate = TRUE, chunk_size = 1e+07,
  control = list(), refit = 1, itnmax = 5000, scale = NULL,
  run = TRUE, burnin = 100, iter = 1000, use.admb = FALSE,
  use.tmb = FALSE, crossed = NULL, reml = FALSE, compile = FALSE,
  extra.args = NULL, strata.labels = NULL, clean = NULL,
  save.matrices = TRUE, simplify = FALSE, getreals = FALSE,
  check = FALSE, prior = FALSE, prior.list = NULL, ...)
```

Arguments

data	Either the raw data which is a dataframe with at least one column named ch (a character field containing the capture history) or a processed dataframe
ddl	Design data list which contains a list element for each parameter type; if NULL it is created
begin.time	Time of first capture(release) occasion
model	Type of c-r model (eg, "cjs", "js")
title	Optional title; not used at present
model.parameters	List of model parameter specifications
design.parameters	Specification of any grouping variables for design data for each parameter
initial	Optional vector of initial values for beta parameters; if named from previous analysis only relevant values are used
groups	Vector of names factor variables for creating groups
time.intervals	Intervals of time between the capture occasions
debug	if TRUE, shows optimization output
method	optimization method for function optimx
hessian	if TRUE, computes v-c matrix using hessian
accumulate	if TRUE, like capture-histories are accumulated to reduce computation

chunk_size	specifies amount of memory to use in accumulating capture histories and design matrices; amount used is 8*chunk_size/1e6 MB (default 80MB)
control	control string for optimization functions
refit	non-zero entry to refit
itnmax	maximum number of iterations for optimization
scale	vector of scale values for parameters
run	if TRUE, it runs model; otherwise if FALSE can be used to test model build components
burnin	number of iterations for mcmc burnin; specified default not realistic for actual use
iter	number of iterations after burnin for mcmc (not realistic default)
use.admb	if TRUE creates data file for cjs.tpl and runs admb optimizer
use.tmb	if TRUE runs TMB for cjs
crossed	if TRUE it uses cjs.tpl or cjs_reml.tpl if reml=FALSE or TRUE respectively; if FALSE, then it uses cjsre which can use Gauss-Hermite integration
reml	if TRUE uses restricted maximum likelihood
compile	if TRUE forces re-compilation of tpl file
extra.args	optional character string that is passed to admb if use.admb==TRUE
strata.labels	labels for strata used in capture history; they are converted to numeric in the order listed. Only needed to specify unobserved strata. For any unobserved strata p=0..
clean	if TRUE, deletes the tpl and executable files for amdb if use.admb=T
save.matrices	for HMM models this option controls whether the gamma,dmat and delta matrices are saved in the model object
simplify	if TRUE, design matrix is simplified to unique valus including fixed values
getreals	if TRUE, compute real values and std errors for TMB models; may want to set as FALSE until model selection is complete
check	if TRUE values of gamma, dmat and delta are checked to make sure the values are valid with initial parameter values.
prior	if TRUE will expect vectors of prior values in list prior.list; currently only implemented for cjsre_tmb
prior.list	which contains list of prior parameters that will be model dependent
...	optional arguments passed to js or cjs and optimx

Details

This function is operationally similar to the function `mark in RMark` in that is is a shell that calls several other functions to perform the following steps: 1) `process.data` to setup data and parameters and package them into a list (processed data),2) `make.design.data` to create the design data for each parameter in the specified model, 3) `create.dm` to create the design matrices for each parameter based on the formula provided for each parameter, 4) call to the specific function for model fitting (now either `cjs_admb` or `js`). As with `mark` the calling arguments for `crm` are

a compilation of the calling arguments for each of the functions it calls (with some arguments renamed to avoid conflicts).#' expects to find a value for ddl. Likewise, if the data have not been processed, then ddl should be NULL. This dual calling structure allows either a single call approach for each model or alternatively for the data to be processed and the design data (ddl) to be created once and then a whole series of models can be analyzed without repeating those steps.

There are some optional arguments that can be used to set initial values and control other aspects of the optimization. The optimization is done with the R package/function optimx and the arguments method and hessian are described with the help for that function. In addition, any arguments not matching those for cjs_admb (the ...) are passed to optimx allowing any of the other parameters to be set. If you set debug=TRUE, then at each function evaluation (cjs.lnl the current values of the parameters and -2*log-likelihood value are output.

In the current implementation, a logit link is used to constrain the parameters in the unit interval (0,1) except for probability of entry which uses an mlogit and N which uses a log link. For the probitCJS model, a probit link is used for the parameters. These could be generalized to use other link functions. Following the notation of MARK, the parameters in the link space are referred to as beta and those in the actual parameter space of Phi and p as reals.

Initial values can be set in 2 ways. To set a baseline intial value for the intercept of Phi p set those arguments to some real value in the open interval (0,1). All non-intercept beta parameters are set to zero. Alternatively, you can specify in initial, a vector of initial values for the beta parameters (on the logit scale). This is most easily done by passing the results from a previous model run using the result list element beta as described below. The code will match the names of the current design matrix to the names in beta and use the appropriate initial values. Any non-specified values are set to 0. If there are no names associated with the initial vector then they are simply used in the specified order. If you do not specify initial values it is equivalent to setting Phi and p to 0.5.

If you have a study with unequal time intervals between capture occasions, then these can be specified with the argument time.intervals.

The argument accumulate defaults to TRUE. When it is TRUE it will accumulate common capture histories that also have common design and common fixed values (see below) for the parameters. This will speed up the analysis because in the calculation of the likelihood (cjs.lnl it loops over the unique values. In general the default will be the best unless you have many capture histories and are using many individual covariate(s) in the formula that would make each entry unique. In that case there will be no effect of accumulation but the code will still try to accumulate. In that particular case by setting accumulate=FALSE you can skip the code run for accumulation.

Most of the arguments controlling the fitted model are contained in lists in the arguments model.parameters and design.parameters which are similar to their counterparts in mark inb RMark. Each is a named list with the names being the parameters in the model (e.g., Phi and p in "cjs" and "Phi","p","pent","N" in "js"). Each named element is also a list containing various values defining the design data and model for the parameter. The elements of model.parameters can include formula which is an R formula to create the design matrix for the parameter and fixed is a matrix of fixed values as described below. The elements of design.parameters can include time.varying, fields, time.bins,age.bins, and cohort.bins. See create.dmdf for a description of the first 2 and create.dm for a description of the last 3.

Real parameters can be set to fixed values using fixed=x where x is a matrix with 3 columns and any number of rows. The first column specifies the particular animal (capture history) as the row number in the dataframe x. The second specifies the capture occasion number for the real parameter to be fixed. For Phi and pent these are 1 to nocc-1 and for p they are 2 to nocc for "cjs" and 1 to

nocc for "js". This difference is due to the parameter labeling by the beginning of the interval for Phi (e.g., survival from occasion 1 to 2) and by the occasion for p. For "cjs" p is not estimated for occasion 1. The third element in the row is the real value in the closed unit interval [0,1] for the fixed parameter. This approach is completely general allowing you to fix a particular real parameter for a specific animal and occasion but it is a bit kludgy. Alternatively, you can set fixed values by specifying values for a field called fix in the design data for a parameter. If the value of fix is NA the parameter is estimated and if it is not NA then the real parameter is fixed at that value. If you also specify fixed as decribed above, they will over-ride any values you have also set with fix in the design data. To set all of the real values for a particular occasion you can use the following example with the dipper data as a template:

```
model.parameters=list(Phi=list(formula=~1, fixed=cbind(1:nrow(dipper),rep(2,nrow(dipper)),rep(1,nrow
```

The above sets Phi to 1 for the interval between occasions 2 and 3 for all animals.

Alternatively, you could do as follows:

data(dipper) dp=process.data(dipper) ddl=make.design.data(dp) ddlPhifix=ifelse(ddlPhitime==2,1,NA)

At present there is no modification of the parameter count to address fixing of real parameters except that if by fixing reals, a beta is not needed in the design it will be dropped. For example, if you were to use ~time for Phi with survival fixed to 1 for time 2, then then beta for that time would not be included.

To use ADMB (use.admb=TRUE), you need to install: 1) the R package R2admb, 2) ADMB, and 3) a C++ compiler (I recommend gcc compiler). The following are instructions for installation with Windows. For other operating systems see (http://www.admb-project.org/downloads) and (http://www.admb-project.org/tools/gcc/).

Windows Instructions:

1) In R use install.packages function or choose Packages/Install Packages from menu and select R2admb.

2) Install ADMB 11: http://www.admb-project.org/downloads. Put the software in C:/admb to avoid problems with spaces in directory name and for the function below to work.

3) Install gcc compiler from: http://www.admb-project.org/tools/gcc/. Put in c:/MinGW

I use the following function in R to setup R2admb to access ADMB rather than adding to my path so gcc versions with Rtools don't conflict.

```
prepare_admb=function()
{
  Sys.setenv(PATH = paste("c:/admb/bin;c:admb/utilities;c:/MinGW/bin;",
        Sys.getenv("PATH"), sep = ";"))
    Sys.setenv(ADMB_HOME = "c:/admb")
    invisible()
}
```

To use different locations you'll need to change the values used above

Before running crm with use.admb=T, execute the function prepare_admb(). You could put this function or the code it contains in your .First or .Rprofile so it runs each time you start R.

Value

crm model object with class=("crm",submodel) where submodel is either "CJS" or "JS" at present.

Author(s)

Jeff Laake

See Also

cjs_admb, js, make.design.data,process.data

Examples

```
{
# cormack-jolly-seber model
# fit 3 cjs models with crm
data(dipper)
dipper.proc=process.data(dipper,model="cjs",begin.time=1)
dipper.ddl=make.design.data(dipper.proc)
mod.Phit.pt=crm(dipper.proc,dipper.ddl,
   model.parameters=list(Phi=list(formula=~time),p=list(formula=~time)))
mod.Phit.pt
mod.Phisex.pdot=crm(dipper.proc,dipper.ddl,groups="sex",
   model.parameters=list(Phi=list(formula=~sex),p=list(formula=~1)))
mod.Phisex.pdot
## if you have RMark installed you can use this code to run the same models
## by removing the comment symbol
#library(RMark)
#data(dipper)
#mod0=mark(dipper,
#model.parameters=list(Phi=list(formula=~time),p=list(formula=~time)),output=FALSE)
#summary(mod0,brief=TRUE)
#mod1=mark(dipper,
#model.parameters=list(Phi=list(formula=~1),p=list(formula=~1)),output=FALSE)
#summary(mod1,brief=TRUE)
#mod2<-mark(dipper,groups="sex",
#model.parameters=list(Phi=list(formula=~sex),p=list(formula=~1)),output=FALSE)
#summary(mod2,brief=TRUE)
# jolly seber model
crm(dipper,model="js",groups="sex",
   model.parameters=list(pent=list(formula=~sex),N=list(formula=~sex)),accumulate=FALSE)

# This example is excluded from testing to reduce package check time
# if you have RMark installed you can use this code to run the same models
# by removing the comment
#data(dipper)
#data(mstrata)
#mark(dipper,model.parameters=list(p=list(formula=~time)),output=FALSE)$results$beta
#mark(mstrata,model="Multistrata",model.parameters=list(p=list(formula=~1),
# S=list(formula=~1),Psi=list(formula=~-1+stratum:tostratum)),
# output=FALSE)$results$beta
#mod=mark(dipper,model="POPAN",groups="sex",
#   model.parameters=list(pent=list(formula=~sex),N=list(formula=~sex)))
#summary(mod)
#CJS example with hmm
crm(dipper,model="hmmCJS",model.parameters = list(p = list(formula = ~time)))
```

```
##MSCJS example with hmm
data(mstrata)
ms=process.data(mstrata,model="hmmMSCJS",strata.labels=c("A","B","C"))
ms.ddl=make.design.data(ms)
ms.ddl$Psi$fix=NA
ms.ddl$Psi$fix[ms.ddl$Psi$stratum==ms.ddl$Psi$tostratum]=1
crm(ms,ms.ddl,model.parameters=list(Psi=list(formula=~-1+stratum:tostratum)))

}
```

`crm.wrapper` *Automation of model runs*

Description

Some functions that help automate running a set of crm models based on parameter specifications.

Usage

```
crm.wrapper(model.list,data,ddl=NULL,models=NULL,base="",
              external=TRUE,run=TRUE,env=NULL,...)

        create.model.list(parameters)

        model.table(model.list)

        load.model(x)

        crmlist_fromfiles(filenames=NULL,external=TRUE)

      rerun_crm(data,ddl,model.list,method=NULL,modelnums=NULL,initial=NULL,...)
```

Arguments

`model.list`	matrix of model names contained in the environment of models function; each row is a model and each column is for a parameter and the value is formula name
`data`	Either the raw data which is a dataframe with at least one column named ch (a character field containing the capture history) or a processed dataframe. For rerun_crm this should be the processed dataframe
`ddl`	Design data list which contains a list element for each parameter type; if NULL it is created; For rerun_crm, must be the same ddl as used with original run can cannot be NULL
`models`	a function with a defined environment with model specifications as variables; values of model.list are some or all of those variables
`base`	base value for model names
`external`	if TRUE, model results are stored externally; otherwise they are stored in crmlist

run	if TRUE, fit models; otherwise just create dml to test if model data are correct for formula
env	environment to find model specifications if not parent.frame
...	aditional arguments passed to crm; for rerun_crm can be used to set hessian=TRUE for specific models after they have been run
parameters	character vector of parameter names
x	filename of externally stored model
method	vector of methods to use for optimization if different that previous run in re-run_crm
modelnums	model numbers to be re-run instead of those that did not covnerge
initial	either a fitted crm model or the model number in model.list to use for starting values
filenames	for non-Windows machine, vector of filenames for external files must be specifed in crmlist_fromfiles including .rda extension

Details

create.model.list creates all combinations of model specifications for the specified set of parameters. In the calling environment it looks for objects named parameter.xxxxxx where xxxxxx can be anything. It creates a matrix with a column for each parameter and as many rows needed to create all combinations. This can be used as input to crm.wrapper.

crm.wrapper runs a sequence of crm models by constructing the call with the arguments and the parameter specifications. The parameter specifications can either be in the local environment or in the environment of the named function models. The advantage of the latter is that it is self-contained such that sets of parameter specifications can be selected without possibility of being over-written or accidentally changed whereas with the former the set must be identified via a script and any in the environment will be used which requires removing/recreating the set to be used.

Value

create.model.list returns a matrix for crm.wrapper; crm.wrapper runs and stores models externally and retrurns a list of model results and a model selection table; load.model returns model object that is stored externally

Author(s)

Jeff Laake

See Also

crm

`deriv_inverse.link` *Derivatives of inverse of link function (internal use)*

Description

Computes derivatives of inverse of link functions (real estimates) with respect to the beta parameters which are used for delta method computation of the var-cov matrix of real parameters.

Usage

```
deriv_inverse.link(real, x, link)
```

Arguments

`real`	Vector of values of real parameters
`x`	Matrix of design values
`link`	Type of link function (e.g., "logit")

Details

Note: that function was renamed to deriv_inverse.link to avoid S3 generic class conflicts. The derivatives of the inverse of the link functions are simple computations using the real values and the design matrix values. The body of the function is as follows:

```
switch(link, logit=x*real*(1-real), log=x*real,
loglog=-real*x*log(real), cloglog=-log(1-real)*x*(1-real), identity=x,
mlogit=x*real*(1-real))
```

Value

Vector of derivative values computed at values of real parameters

Author(s)

Jeff Laake

See Also

`inverse.link`, `compute.real`

`dipper` *Dipper capture-recapture data*

Description

A capture-recapture data set on European dippers from France that accompanies MARK as an example analysis using the CJS and POPAN models. The dipper data set was orginally described as an example by Lebreton et al (1992).

Format

A data frame with 294 observations on the following 2 variables.

ch a character vector containing the encounter history of each bird

sex the sex of the bird: a factor with levels `Female` `Male`

Details

This is a data set that accompanies program MARK as an example for CJS and POPAN analyses. The data can be stratified using sex as a grouping variable. The functions `run.dipper`, `run.dipper.alternate`, `run.dipper.popan` defined below in the examples mimic the models used in the dbf file that accompanies MARK. Note that the models used in the MARK example use PIM coding with the sin link function which is often better at identifying the number of estimable parameters. The approach used in the R code uses design matrices and cannot use the sin link and is less capable at counting parameters. These differences are illustrated by comparing the results of `run.dipper` and `run.dipper.alternate` which fit the same set of "CJS" models. The latter fits the models with constraints on some parameters to achieve identifiability and the former does not. Although it does not influence the selection of the best model it does infleunce parameter counts and AIC ordering of some of the less competitive models. In using design matrices it is best to constrain parameters that are confounded (e.g., last occasion parameters in Phi(t)p(t) CJS model) when possible to achieve more reliable counts of the number of estimable parameters.

Note that the covariate "sex" defined in dipper has values "Male" and "Female". It cannot be used directly in a formula for MARK without using it do define groups because MARK.EXE will be unable to read in a covariate with non-numeric values. By using `groups="sex"` in the call the `process.data` a factor "sex" field is created that can be used in the formula. Alternatively, a new covariate could be defined in the data with say values 0 for Female and 1 for Male and this could be used without defining groups because it is numeric. This can be done easily by translating the values of the coded variables to a numeric variable. Factor variables are numbered 1..k for k levels in alphabetic order. Since Female < Male in alphabetic order then it is level 1 and Male is level 2. So the following will create a numeric sex covariate.

```
dipper$numeric.sex=as.numeric(dipper$sex)-1
```

Source

Lebreton, J.-D., K. P. Burnham, J. Clobert, and D. R. Anderson. 1992. Modeling survival and testing biological hypotheses using marked animals: case studies and recent advances. Ecol. Monogr. 62:67-118.

dmat_hsmm2hmm *Create expanded state-dependent observation matrix for HMM from HSMM*

Description

Creates expanded state-dependent matrix for HMM from aggregated state-dependent observation matrix.

Usage

```
dmat_hsmm2hmm(dmat, mv)
```

Arguments

dmat	state-dependent observation matrix for aggregated states
mv	vector of dwell time distribution lengths

Value

expanded state-dependent observation matrix for hmm to approximate hsmm

Author(s)

Jeff Laake

fix.parameters *Fixing real parameters in crm models*

Description

Creates matrix with appropriate structure for fixing real parameters in a crm model for a specified set of capture histories and occasions.

Usage

```
fix.parameters(x, subset = NULL, occasions, values)
```

Arguments

x	processed data list
subset	logical expression for set of animals; used in subset function
occasions	vector of occasion numbers
values	either single or vector of values; if latter, must match length of occasions

Value

Matrix with 3 columns: 1) animal row number, 2) occasion, 3) value for real parameter

Author(s)

Jeff Laake

`function.wrapper` *Utility extract functions*

Description

Several functions have been added to extract components from externally saved crm models function.wrapper accepts a character string for a model, loads it and then runs the specified function on it. Currently, only 2 functions have been defined: fx.aic to compute aic and fx.par.count to extract parameter count. Currently parameters other than x (eg chat) are passed through the environment. Possibly could have used ...

Usage

```
function.wrapper(x,fx,base="",...)

       fx.aic(x)

       fx.par.count(x)
```

Arguments

`x`	character string for the model stored as external object (.rda)
`fx`	function to be called for each model
`base`	base name for models
`...`	additional values that are added to environment of fx (eg chat for fx.aic)

Value

extracted value defined by function

Author(s)

Jeff Laake

`global_decode` *Global decoding of HMM*

Description

Computes sequence of state predictions for each individual

Usage

```
global_decode(object, ddl = NULL, state.names = NULL)
```

Arguments

`object`	fitted crm model (must be an HMM model)
`ddl`	design data list; will be computed if NULL
`state.names`	names for states used to label output; if NULL uses strata.labels + Dead state

Value

matrix of state predictions

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. See page 82.

Examples

```
#

# This example is excluded from testing to reduce package check time
# cormack-jolly-seber model
data(dipper)
mod=crm(dipper,model="hmmcjs")
global_decode(mod)
```

hmmDemo *HMM computation demo functions*

Description

Uses fitted hmm model to construct HMM state vectors alpha and phi for demonstration purposes

Usage

```
hmmDemo(object, ddl = NULL, state.names = NULL, obs.names = NULL)
```

Arguments

object	fitted hmm model
ddl	design dat list; if NULL it is created
state.names	names for states used to label output; if NULL uses strata.labels + Dead state
obs.names	names for observations used to label output; if NULL uses ObsLevels

Value

hmm demo list which includes 1) lnl - the log-likelihood value, 2) alpha - forward probabilities, 3) beta - backward probabilities, 4) phi - scaled forward probabilities, 5) v- intermediate calculation for phi, 6) dmat - 3-d array with observation probability matrix for each occasion, 7) gamma - 3-d array with state transition probability matrix for each occasion, 8) stateprob - predicted state probabilities, 9) local_decode - state predictions for each occasion and individual, 10) global_decode - state predictions for entire sequence for each individual.

Author(s)

Jeff Laake

Examples

```
# This example is excluded from testing to reduce package check time
# cormack-jolly-seber model
data(dipper)
mod=crm(dipper,model="hmmcjs")
x=hmmDemo(mod,state.names=c("Alive","Dead"),obs.names=c("Missed","Seen"))
par(mfrow=c(2,1))
barplot(t(x$alpha[45,,]),beside=TRUE,names.arg=x$chforwardstrings)
barplot(t(x$phi[45,,]),beside=TRUE,names.arg=x$chforwardstrings)
# multi-state example showing state predictions
data(mstrata)
mod=crm(mstrata,model="hmmMSCJS",strata.labels=c("A","B","C"))
#' x=hmmDemo(mod)
# state predictions are normalized by likelihood value which = rowSums(alpha*beta)
cat(paste("\nrowsums = ",rowSums(x$alpha[45,,]*x$beta[45,,],na.rm=TRUE)[2],
```

```
   "which matches likelihood value",exp(x$lnl[45]),"\n"))
# state predictions given the data
x$stateprob[45,,]
```

HMMLikelihood *Hidden Markov Model likelihood functions*

Description

Function HMMLikelihood computes the log-likelihood via hmm.lnl which is a wrapper for the FORTRAN code hmm_like.f. The function HMMlikelihood is called from optimizer and it in turn calls hmm.lnl after setting up parameters.

For an R version of the HMMLikelihood and related code see R_HMMLikelihood

Usage

```
HMMLikelihood(par,type,xx,xstart,mx,T,freq=1,fct_dmat,fct_gamma,fct_delta,ddl,
                    dml,parameters,debug=FALSE,return.mat=FALSE,sup=NULL,check=FALSE)
       reals(ddl,dml,parameters,parlist,indices=NULL)
       hmm.lnl(x,start,m,T,dmat,gamma,delta,freq,debug)
```

Arguments

par	vector of parameter values for log-likelihood evaluation
type	vector of parameter names used to split par vector into types
xx	matrix of observed sequences (row:id; column:occasion/time); xx used instead of x to avoid conflict in optimx
xstart	for each ch, the first non-zero x value and the occasion of the first non-zero value; ; xstart used instead of start to avoid conflict in optimx
mx	number of states; mx used instead of m to avoid conflict in optimx
T	number of occasions; sequence length
freq	vector of history frequencies or 1
fct_dmat	function to create D from parameters
fct_gamma	function to create gamma - transition matrix
fct_delta	function to create initial state distribution
ddl	design data list of parameters for each id
dml	list of design matrices; one entry for each parameter; each entry contains fe and re for fixed and random effects
parameters	formulas for each parameter type
debug	if TRUE, print out par values and -log-likelihood
return.mat	If TRUE, returns list of transition, observation and delta arrays.

sup	list of supplemental information that may be needed by the function but only needs to be computed once; currently only used for MVMS models for dmat
check	if TRUE, checks validity of gamma, dmat and delta to look for any errors
x	same as xx but for call to hmm.lnl
m	same as mx but for call to hmm.lnl
dmat	observation probability matrices
gamma	transition matrices
delta	initial distribution
parlist	list of parameter strings used to split par vector
start	same as xstart but for hmm.lnl
indices	specific indices for computation unless NULL

Value

HMMLikelihood returns log-likelihood for a single sequence and hmm.lnl returns the negative log-likelihood value for each capture history. reals returns either the column dimension of design matrix for parameter or the real parameter vector

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. 275p.

See Also

R_HMMLikelihood

---

hsmm2hmm *Compute transition matrix for HMM from HSMM*

Description

Computes expanded transition matrix for HMM from aggregated state transition matrix and dwell time distributions.

Usage

```
hsmm2hmm(omega, dm, eps = 1e-10)
```

Arguments

omega	transition matrix for aggregated states
dm	list of dwell time distribution vectors; each list element is vector of dwell time probabilities in order of aggregated states
eps	epsilon value for setting small probabilities to 0; anything less than eps is set to exactly 0.

Value

transition matrix for hmm to approximate hsmm

Author(s)

Walter Zucchini

---

initiate_pi *Setup fixed values for pi in design data*

Description

Creates field fix in pi design data and either sets to NA to be estimated or 1 if intercept or 0 if not possible. It uses observed data at initial entry to decide the appropriate fix values.

Usage

```
initiate_pi(data, ddl)
```

Arguments

data	Processed data list; resulting value from process.data
ddl	design data list created by make.design.data

Details

It is possible that the state will be known for some individuals and unknown for others. The function initiate_pi modifies the design data for pi by adding the field fix which can be used to fix the value of pi for an individual with known state at release. It also sets up the mlogit structure for pi by setting one of the fixed values to 1 such that the value for pi is computed by subtraction. By default it chooses the first factor level of each of the state variables that is unknown. If some of the state variables are known but others are not, it sets fix to 0 for the values of the known state variables that don't match the known value.

For example, with a bivariate case with first state A,B or C and second variable 1 or 2. Only the second variable can be unknown. There are 6 possible combinations of the state variables: A1,A2,B2,B2,C1,C2. Assume a capture history is 0,Au,0,B1,B2. There will be 6 records in the design data for this record with strata A1,A2,B2,B2,C1,C2. The occ (occasion field) will be 2 because pi is only for the first occasion. After running initiate_pi, the fix values will be 1,NA,0,0,0,0.

A1 will be the intercept and computed by subtraction (fix=1), the value for A2 will be estimated (NA) and all of the others are not possible (0) because it is known to be in A.

If both can be uncertain, and the capture history was 0,uu,0,B1,B2, then the fix values will be 1,NA,NA,NA,NA,NA. With a mix of partially known and completely unknown the formulation for the mlogit could get complicated making it difficult to specify an understandable formula because the restrictions on the mlogit will change, so think hard and long about what you are doing. It may be necessary to construct "multiple" formula in the sense that you use indicator variables (0/1) as interactions.

Value

ddl with fix field added to pi dataframe

Author(s)

Jeff Laake

`inverse.link` *Inverse link functions (internal use)*

Description

Computes values of inverse of link functions for real estimates.

Usage

```
inverse.link(x, link)
```

Arguments

`x`	Matrix of design values multiplied by the vector of the beta parameter values
`link`	Type of link function (e.g., "logit")

Details

The inverse of the link function is the real parameter value. They are simple functions of `X*Beta` where `X` is the design matrix values and `Beta` is the vector of link function parameters. The body of the function is as follows:

```
switch(link, logit=exp(x)/(1+exp(x)), log=exp(x),
loglog=exp(-exp(-x)), cloglog=1-exp(-exp(x)), identity=x,
mlogit=exp(x)/(1+sum(exp(x))) )
```

The `link="mlogit"` only works if the set of real parameters are limited to those within the set of parameters with that specific link. For example, in POPAN, the `pent` parameters are of type "mlogit" so the probabilities sum to 1. However, if there are several groups then each group will have a different set of `pent` parameters which are identified by a different grouping of the "mlogit"

parameters (i.e., "mlogit(1)" for group 1, "mlogit(2)" for group 2 etc). Thus, in computing real parameter values (see `compute.real`) which may have varying links, those with "mlogit" are not used with this function using `link="mlogit"`. Instead, the link is temporarily altered to be of type "log" (i.e., inverse=exp(x)) and then summed over sets with a common value for "mlogit(j)" to construct the inverse for "mlogit" as `exp(x)/(1+sum(exp(x))`.

Value

Vector of real values computed from `x=X*Beta`

Author(s)

Jeff Laake

See Also

`compute.real`,`deriv_inverse.link`

`js` *Fitting function for Jolly-Seber model using Schwarz-Arnason POPAN formulation*

Description

A function for computing MLEs for a specified Jolly-Seber open population capture-recapture model for processed dataframe `x` with user specified formulas in `parameters` that create list of design matrices `dml`. This function can be called directly but is most easily called from `crm` that sets up needed arguments.

Usage

```
js(x, ddl, dml, model_data = NULL, parameters, accumulate = TRUE,
  initial = NULL, method = "BFGS", hessian = FALSE, debug = FALSE,
  chunk_size = 1e+07, refit, itnmax = NULL, control = NULL, scale, ...)
```

Arguments

`x`	processed dataframe created by process.data
`ddl`	list of dataframes for design data; created by call to `make.design.data`
`dml`	list of design matrices created by `create.dm` from formula and design data
`model_data`	a list of all the relevant data for fitting the model including imat, Phi.dm,p.dm,Phi.fixed,p.fixed, and time.intervals. It is used to save values and avoid accumulation again if the model was re-rerun with an additional call to js when using autoscale or re-starting with initial values. It is stored with returned model object.
`parameters`	equivalent to `model.parameters` in `crm`
`accumulate`	if TRUE will accumulate capture histories with common value and with a common design matrix for Phi and p to speed up execution

initial	initial values for parameters if desired; if named vector from previous run it will match to columns with same name
method	method to use for optimization; see optimx
hessian	if TRUE will compute and return the hessian
debug	if TRUE will print out information for each iteration
chunk_size	specifies amount of memory to use in accumulating capture histories; amount used is 8*chunk_size/1e6 MB (default 80MB)
refit	non-zero entry to refit
itnmax	maximum number of iterations
control	control string for optimization functions
scale	vector of scale values for parameters
...	any remaining arguments are passed to additional parameters passed to optimx or js.lnl

Details

It is easiest to call js through the function crm. Details are explained there.

Be cautious with this function at present. It does not include many checks to make sure values like fixed values will remain in the specified range of the data. Normally this would not be a big problem but because js.lnl calls an external FORTRAN subroutine via cjs.lnl, if it gets a subscirpt out of bounds, it will cause R to terminate. So make sure to save your workspace frequently if you use this function in its current implementation.

Value

The resulting value of the function is a list with the class of crm,js such that the generic functions print and coef can be used.

beta	named vector of parameter estimates
lnl	-2*log likelihood
AIC	lnl + 2* number of parameters
convergence	result from optimx; if 0 optimx thinks it converged
count	optimx results of number of function evaluations
reals	dataframe of data and real Phi and p estimates for each animal-occasion excluding those that occurred before release
vcv	var-cov matrix of betas if hessian=TRUE was set

Author(s)

Jeff Laake

References

Schwarz, C. J., and A. N. Arnason. 1996. A general methodology for the analysis of capture-recapture experiments in open populations. Biometrics 52:860-873.

js.accumulate *Accumulates common capture history values*

Description

To speed up compuation, animals with the same capture history and design matrix are accumulated and represented by a frequency. Computes starting values for Phi and p parameters from the list of design matrices and the summarized data list including ch matrix and first and last vectors. If any values are missing (NA) or abs(par)>5, they are set to 0.

Usage

```
js.accumulate(x, model_data, nocc, freq, chunk_size)
```

Arguments

x	data
model_data	list of design matrices, fixed parameters and time intervals all which can vary by animal
nocc	number of capture occasions
freq	frequency of each capture history before accumulation
chunk_size	size that determines number of pieces of data/design matrix that are handled. Smaller chunk_size requires more time but less memory. 1e7 is default set in cjs.

Value

modified model_data list that is accumulated

Author(s)

Jeff Laake

js.hessian *Compute variance-covariance matrix for fitted JS model*

Description

A wrapper function that sets up call to hessian function to compute and then invert the hessian.

Usage

```
js.hessian(model)
```

Arguments

model	fitted JS model from function crm

Value

variance-covariance matrix for specified model or the model object with the stored vcv depending on whether the model has already been run

Author(s)

Jeff Laake

---

js.lnl *Likelihood function for Jolly-Seber model using Schwarz-Arnason POPAN formulation*

Description

For a given set of parameters and data, it computes -2*log Likelihood value but does not include data factorials. Factorials for unmarked are not needed but are included in final result by js so the result matches output from MARK for the POPAN model.

Usage

```
js.lnl(par, model_data, debug = FALSE, nobstot, jsenv)
```

Arguments

par	vector of parameter values
model_data	a list that contains: 1)imat-list of vectors and matrices constructed by process.ch from the capture history data, 2)Phi.dm design matrix for Phi constructed by create.dm, 3)p.dm design matrix for p constructed by create.dm, 4)pent.dm design matrix for probability of entry constructed by create.dm, 5) N.dm design matrix for estimates of number of animals not caught from super-population constructed by create.dm, 6)Phi.fixed matrix with 3 columns: ch number(i), occasion number(j), fixed value(f) to fix phi(i,j)=f, 7) p.fixed matrix with 3 columns: ch number(i), occasion number(j), 8) pent.fixed matrix with 3 columns: ch number(i), occasion number(j), fixed value(f) to fix pent(i,j)=f, and 9) time.intervals intervals of time between occasions if not all 1 fixed value(f) to fix p(i,j)=f
debug	if TRUE will printout values of par and function value
nobstot	number of unique caught at least once by group if applicable
jsenv	environment for js to update iteration counter

Details

This functions uses cjs.lnl and then supplements with the remaining calculations to compute the likelihood for the POPAN formulation (Arnason and Schwarz 1996) of the Jolly-Seber model.

Value

-log likelihood value, excluding data (ui) factorials which are added in js after optimization to match MARK

Author(s)

Jeff Laake

References

Schwarz, C. J., and A. N. Arnason. 1996. A general methodology for the analysis of capture-recapture experiments in open populations. Biometrics 52:860-873.

---

local_decode *Local decoding of HMM*

Description

Computes state predictions one at a time for each occasion for each individual

Usage

```
local_decode(object, ddl = NULL, state.names = NULL)
```

Arguments

object	fitted crm model (must be an HMM model)
ddl	design data list
state.names	names for states used to label output; if NULL uses strata.labels + Dead state

Value

matrix of state predictions

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. See page 80.

Examples

```
#

# This example is excluded from testing to reduce package check time
# cormack-jolly-seber model
data(dipper)
mod=crm(dipper,model="hmmcjs")
local_decode(mod)
```

`make.design.data` *Create design dataframes for crm*

Description

For each type of parameter in the analysis model (e.g, p, Phi), this function creates design data for model fitting from the original data. These design data expand the original data record for a single(freq=1)/multiple(freq>1) individuals to many records where each record is for a single occasion. The design data can be specific to the parameter and a list of design data frames are returned with a dataframe for each parameter.

Usage

```
make.design.data(data, parameters = list())
```

Arguments

`data`	Processed data list; resulting value from process.data
`parameters`	Optional list containing a list for each type of parameter (list of lists); each parameter list is named with the parameter name (eg Phi); each parameter list can contain vectors named age.bins,time.bins and cohort.bins, time.varying, and static

`age.bins`	bins for binning ages
`time.bins`	bins for binning times
`cohort.bins`	bins for binning cohorts
`time.varying`	vector of field names that are time varying for this parameter
`static`	vector of field names that are to be included that are not time varying for this parameter

Details

For each record in the design data, default data fields are created based on the model. For example, for CJS, the default fields include cohort, age, time which are factor variables and Cohort, Age, and Time which are numeric versions of the same fields. The values of these can be altered by values of begin.time, time.intervals and initial.ages set in the call to `process.data`. In addition if groups are identifed the grouping factor variables are added to the design data. Any other fields in the data

are repeated on each record for the animal(s) (eg weight), unless they are defined as time.varying in which case the fields should be named with the convention xn where x is the base name for the variable and n is the time value (eg, td1999, td2000,...). For time varying fields the variable name in the design data is the base name and the value for the record is the one for that occasion(time). The variables can be refined for each parameter by including argument static with the character vector of static fields to include.

After creating design data, you can create a field in the dataframe for a parameter named fix and it can be assigned values at the real parameter for that occasion/id will be fixed at the value. For parameters that are not supposed to be fixed, the field fix should be assigned NA. For example, ddlPhifix=ifelse(ddlPhitime==2,1,NA) will assign all real Phi values to 1 for the interval beginning at time 2. If there is no field fix, then no parameters are fixed. For mlogit parameters, a fix field is added automatically and the value 1 is specified for the stratum that is supposed to be computed by subtraction and the remainder are set to NA. If some Psi are not possible then those values can be changed to 0.

The following variable names are reserved and should not be used in the data: occ,age,time,cohort,Age,Time,Cohort,Y,Z,initia

Value

The function value is a list of data frames. The list contains a data frame for each type of parameter in the model (e.g., Phi and p for CJS). The names of the list elements are the parameter names (e.g., Phi). The structure of the dataframe depends on the calling arguments and the model & data structure. In addition the value of parameters argument is saved as design.parameters.

Author(s)

Jeff Laake

See Also

process.data,merge_design.covariates

Examples

```
# This example is excluded from testing to reduce package check time
data(dipper)
dipper.proc=process.data(dipper)
ddl=make.design.data(dipper.proc)
```

`merge_design.covariates` *Merge time (occasion) and/or group specific covariates into design data*

Description

Adds new design data fields from a dataframe into a design data list (ddl) by matching via time and/or group field in the design data.

Usage

```
merge_design.covariates(ddl, df, bygroup = FALSE, bytime = TRUE)
```

Arguments

ddl	current design dataframe for a specific parameter and not the entire design data list (ddl)
df	dataframe with time(occasion) and/or group-specific data
bygroup	logical; if TRUE, then a field named group should be in df and the values can then be group specific.
bytime	logical; if TRUE, then a field named time should be in df and the values can then be time specific.

Details

Design data can be added to the parameter specific design dataframes with R commands. This function simplifies the process by enabling the merging of a dataframe with a time and/or group field and one or more time and/or group specific covariates into the design data list for a specific model parameter. This is a replacement for the older function merge.occasion.data. Unlike the older function, it uses the R function merge but before merging it makes sure all of the fields exist and that you are not merging data that already exists in the design data. It also maintains the row names in the case where design data have been deleted prior to merging the design covariate data.

If bytime=TRUE,the dataframe df must have a field named time that matches 1-1 for each value of time in the design data list (ddl). All fields in df (other than time/group) are added to the design data. If you set bygroup=TRUE and have a field named group in df and its values match the group fields in the design data then group-specific values can be assigned for each time if bytime=TRUE. If bygroup=TRUE and bytime=FALSE then it matches by group and not by time.

Value

Design dataframe (for a particular parameter) with new fields added. See make.design.data for a description of the design data list structure. The return value is only one element in the list rather than the entire list as with the older function merge.occasion.data.

Author(s)

Jeff Laake

See Also

make.design.data, process.data

Examples

```
data(dipper)
dipper.proc=process.data(dipper)
ddl=make.design.data(dipper.proc)
df=data.frame(time=c(1:7),effort=c(10,5,2,8,1,2,3))
# note that the value for time 1 is superfluous for CJS but not for POPAN
# the value 10 will not appear in the summary because there is no p for time 1
summary(ddl$p)
ddl$p=merge_design.covariates(ddl$p,df)
summary(ddl$p)
#Statement below will create an error because a value for time 7 not given
#ddl=merge.occasion.data(dipper.proc,ddl,"p",data.frame(time=c(1:6),effort=c(10,5,2,8,1,2)))
#
# Assign group-specific values
#

data(dipper)
dipper.proc=process.data(dipper)
ddl=make.design.data(dipper.proc)
df=data.frame(time=c(1:7),effort=c(10,5,2,8,1,2,3))
# note that the value for time 1 is superfluous for CJS but not for POPAN
# the value 10 will not appear in the summary because there is no p for time 1
summary(ddl$p)
ddl$p=merge_design.covariates(ddl$p,df)
summary(ddl$p)
#Statement below will create an error because a value for time 7 not given
#ddl=merge.occasion.data(dipper.proc,ddl,"p",data.frame(time=c(1:6),effort=c(10,5,2,8,1,2)))
#
# Assign group-specific values
#
dipper.proc=process.data(dipper,groups="sex")
ddl=make.design.data(dipper.proc)
df=data.frame(group=c(rep("Female",6),rep("Male",6)),time=rep(c(2:7),2),
effort=c(10,5,2,8,1,2,3,20,10,4,16,2))
merge_design.covariates(ddl$p,df,bygroup=TRUE)
```

`mixed.model.admb` *Mixed effect model contstruction*

Description

Functions that develop structures needed for a mixed effect model

Usage

```
mixed.model.admb(formula,data)
```

```
mixed.model(formula,data,indices=FALSE)

mixed.model.dat(x,con,idonly,n)

reindex(x,id)
```

Arguments

formula	formula for mixed effect mode in the form used in lme4; ~fixed +(re1|g1) +...+(ren|gn)
data	dataframe used to construct the design matrices from the formula
x	list structure created by mixed.model.admb
con	connection to data file which contents will be appended
id	vector of factor values used to split the data up by individual capture history
idonly	TRUE, if random effects not crossed
n	number of capture history records
indices	if TRUE, outputs structure with indices into dm for random effects

Details

mixed.model.admb - creates design matrices and supporting index matrices for use of mixed model in ADMB

mixed.model - creates design matrices and supporting index matrices in an alternate list format that is not as easily used in ADMB

mixed.model.dat - writes to data file (con) for fixed and random effect stuctures

reindex - creates indices for random effects that are specific to the individual capture history; it takes re.indices, splits them by id and creates a ragged array by id (used.indices) with the unique values for that id. index.counts is the number of indices per id to read in ragged array. It then changes re.indices to be an index to the indices within the id from 1 to the number of indices within the id.

Value

mixed.model.admb returns a list with elements re.dm, a combined design matrix for all of the random effects; and re.indices, matrix of indices into a single vector of random effects to be applied to the design matrix location. mixed.model returns a list (re.list) with an element for each random effect structure. The contents are a standard design matrix (re.dm) if indices==FALSE and a re.dm and re.indices which matches the structure of mixed.model.admb. mixed.model will be more useful with R than ADMB.

Author(s)

Jeff Laake

mscjs *Fitting function for Multistate CJS models*

Description

A function for computing MLEs for a Multi-state Cormack-Jolly-Seber open population capture-recapture model for processed dataframe x with user specified formulas in parameters that create list of design matrices dml. This function can be called directly but is most easily called from crm that sets up needed arguments.

Usage

```
mscjs(x, ddl, dml, model_data = NULL, parameters, accumulate = TRUE,
  initial = NULL, method, hessian = FALSE, debug = FALSE,
  chunk_size = 1e+07, refit, itnmax = NULL, control = NULL, scale,
  re = FALSE, compile = FALSE, extra.args = "", clean = TRUE, ...)
```

Arguments

x	processed dataframe created by process.data
ddl	list of dataframes for design data; created by call to make.design.data
dml	list of design matrices created by create.dm from formula and design data
model_data	a list of all the relevant data for fitting the model including imat, S.dm,p.dm,Psi.dm,S.fixed,p.fixed,Psi.fixe and time.intervals. It is used to save values and avoid accumulation again if the model was re-rerun with an additional call to cjs when using autoscale or re-starting with initial values. It is stored with returned model object.
parameters	equivalent to model.parameters in crm
accumulate	if TRUE will accumulate capture histories with common value and with a common design matrix for S and p to speed up execution
initial	list of initial values for parameters if desired; if each is a named vector from previous run it will match to columns with same name
method	method to use for optimization; see optim
hessian	if TRUE will compute and return the hessian
debug	if TRUE will print out information for each iteration
chunk_size	specifies amount of memory to use in accumulating capture histories; amount used is 8*chunk_size/1e6 MB (default 80MB)
refit	non-zero entry to refit
itnmax	maximum number of iterations
control	control string for optimization functions
scale	vector of scale values for parameters
re	if TRUE creates random effect model admbcjsre.tpl and runs admb optimizer
compile	if TRUE forces re-compilation of tpl file

extra.args	optional character string that is passed to admb
clean	if TRUE, deletes the tpl and executable files for amdb
...	not currently used

Details

It is easiest to call mscjs through the function crm. Details are explained there.

Value

The resulting value of the function is a list with the class of crm,cjs such that the generic functions print and coef can be used.

beta	named vector of parameter estimates
lnl	-2*log likelihood
AIC	lnl + 2* number of parameters
convergence	result from optim; if 0 optim thinks it converged
count	optim results of number of function evaluations
reals	dataframe of data and real S and p estimates for each animal-occasion excluding those that occurred before release
vcv	var-cov matrix of betas if hessian=TRUE was set

Author(s)

Jeff Laake

References

Ford, J. H., M. V. Bravington, and J. Robbins. 2012. Incorporating individual variability into mark-recapture models. Methods in Ecology and Evolution 3:1047-1054.

Examples

```
# this example requires admb
# The same example is in the RMark package and it is included here to
# illustrate the differences in the handling of mlogit parameters between RMark
# and marked.  The MARK software handles parameters like Psi which must sum to 1
# by excluding one of the cells that is used as a reference cell and is computed by
# subtracting the other cell values from 1 so the total sums to 1.  This is often
# handled with an mlogit parameter in which the cell values are exp(beta) and the
# reference cell is set to 1 and the values are divided by the sum across the cells
# so the resulting values are probabilities that sum to 1. In marked, instead of removing
# one of the cells, all are included and the user must select which should be the
# reference cell by setting the value fix=1 for that cell and others are NA so they are
# estimated. For transition parameters like Psi, the default design data is setup so
# that the probability of remaining in the cell (stratum=tostratum) is the reference cell
# and fix set to 1.  Thus, this means 2 changes are needed to the script in RMark.
# The first is to remove the statement skagit.ddl$Psi$fix=NA because that over-rides
```

```
# the default fix values.  The other is to add
# skagit.ddl$Psi$fix[skagit.ddl$Psi$stratum=="B"&skagit.ddl$Psi$tostratum=="B"&
#  skagit.ddl$Psi$time==5]=0
# to change the value from 1 to 0 which forces movement from B to A in the interval 5 to 6. If
# this is not done then Psi B to B=Psi B to A=0.5 because each is 1 and when they are normalized
# they are divided by the sum which is 2 (1/2).
if(class(try(setup_admb("mscjs")))!="try-error")
{
data(skagit)
skagit.processed=process.data(skagit,model="Mscjs",groups=c("tag"),strata.labels=c("A","B"))
skagit.ddl=make.design.data(skagit.processed)
#
# p
#
# Can't be seen at 5A or 2B,6B (the latter 2 don't exist)
skagit.ddl$p$fix=ifelse((skagit.ddl$p$stratum=="A"&skagit.ddl$p$time==5) |
(skagit.ddl$p$stratum=="B"&skagit.ddl$p$time%in%c(2,6)),0,NA)
# Estimated externally from current data to allow estimation of survival at last interval
skagit.ddl$p$fix[skagit.ddl$p$tag=="v7"&skagit.ddl$p$time==6&skagit.ddl$p$stratum=="A"]=0.687
skagit.ddl$p$fix[skagit.ddl$p$tag=="v9"&skagit.ddl$p$time==6&skagit.ddl$p$stratum=="A"]=0.975
#
# Psi
#
# only 3 possible transitions are A to B at time interval 2 to 3 and
# for time interval 3 to 4 from A to B and from B to A
# rest are fixed values
############ change for RMark to marked; remove next line
#skagit.ddl$Psi$fix=NA
# stay in A for intervals 1-2, 4-5 and 5-6
skagit.ddl$Psi$fix[skagit.ddl$Psi$stratum=="A"&
 skagit.ddl$Psi$tostratum=="B"&skagit.ddl$Psi$time%in%c(1,4,5)]=0
# stay in B for interval 4-5
skagit.ddl$Psi$fix[skagit.ddl$Psi$stratum=="B"&skagit.ddl$Psi$tostratum=="A"
 &skagit.ddl$Psi$time==4]=0
# leave B to go to A for interval 5-6
skagit.ddl$Psi$fix[skagit.ddl$Psi$stratum=="B"&skagit.ddl$Psi$tostratum=="A"&
skagit.ddl$Psi$time==5]=1
############ change for RMark to marked; add next line to set B to B to 0 otherwise it has
############ been set to 1 by default which would make psi B to B = psi B to A = 0.5
skagit.ddl$Psi$fix[skagit.ddl$Psi$stratum=="B"&skagit.ddl$Psi$tostratum=="B"&
skagit.ddl$Psi$time==5]=0
# "stay" in B for interval 1-2 and 2-3 because none will be in B
skagit.ddl$Psi$fix[skagit.ddl$Psi$stratum=="B"&skagit.ddl$Psi$tostratum=="A"&
skagit.ddl$Psi$time%in%1:2]=0
#
# S
#
# None in B, so fixing S to 1
skagit.ddl$S$fix=ifelse(skagit.ddl$S$stratum=="B"&skagit.ddl$S$time%in%c(1,2),1,NA)
skagit.ddl$S$fix[skagit.ddl$S$stratum=="A"&skagit.ddl$S$time==4]=1
# fit model
p.timexstratum.tag=list(formula=~time:stratum+tag,remove.intercept=TRUE)
Psi.sxtime=list(formula=~-1+stratum:time)
```

```
S.stratumxtime=list(formula=~-1+stratum:time)
#
mod1=crm(skagit.processed,skagit.ddl,
model.parameters=list(S=S.stratumxtime,p= p.timexstratum.tag,Psi=Psi.sxtime),hessian=TRUE)
if(class(mod1)[1]!="try-error") mod1
}
```

`mscjs_tmb` *Fitting function for Multistate CJS models with TMB*

Description

A function for computing MLEs for a Multi-state Cormack-Jolly-Seber open population capture-recapture model for processed dataframe `x` with user specified formulas in `parameters` that create list of design matrices `dml`. This function can be called directly but is most easily called from `crm` that sets up needed arguments.

Usage

```
mscjs_tmb(x, ddl, fullddl, dml, model_data = NULL, parameters,
  accumulate = TRUE, initial = NULL, method, hessian = FALSE,
  debug = FALSE, chunk_size = 1e+07, refit, itnmax = NULL,
  control = NULL, scale, re = FALSE, compile = FALSE, extra.args = "",
  clean = TRUE, ...)
```

Arguments

`x`	processed dataframe created by process.data
`ddl`	list of simplified dataframes for design data; created by call to `make.design.data`
`fullddl`	list of complete dataframes for design data; created by call to `make.design.data`
`dml`	list of design matrices created by `create.dm` from formula and design data
`model_data`	a list of all the relevant data for fitting the model including imat, S.dm,p.dm,Psi.dm,S.fixed,p.fixed,Psi.fixe and time.intervals. It is used to save values and avoid accumulation again if the model was re-rerun with an additional call to cjs when using autoscale or re-starting with initial values. It is stored with returned model object.
`parameters`	equivalent to `model.parameters` in `crm`
`accumulate`	if TRUE will accumulate capture histories with common value and with a common design matrix for S and p to speed up execution
`initial`	list of initial values for parameters if desired; if each is a named vector from previous run it will match to columns with same name
`method`	method to use for optimization; see `optim`
`hessian`	if TRUE will compute and return the hessian
`debug`	if TRUE will print out information for each iteration
`chunk_size`	specifies amount of memory to use in accumulating capture histories; amount used is 8*chunk_size/1e6 MB (default 80MB)

refit	non-zero entry to refit
itnmax	maximum number of iterations
control	control string for optimization functions
scale	vector of scale values for parameters
re	if TRUE creates random effect model admbcjsre.tpl and runs admb optimizer
compile	if TRUE forces re-compilation of tpl file
extra.args	optional character string that is passed to admb
clean	if TRUE, deletes the tpl and executable files for amdb
...	not currently used

Details

It is easiest to call mscjs_tmb through the function crm. Details are explained there.

Value

The resulting value of the function is a list with the class of crm,cjs such that the generic functions print and coef can be used.

beta	named vector of parameter estimates
lnl	-2*log likelihood
AIC	lnl + 2* number of parameters
convergence	result from optim; if 0 optim thinks it converged
count	optim results of number of function evaluations
reals	dataframe of data and real S and p estimates for each animal-occasion excluding those that occurred before release
vcv	var-cov matrix of betas if hessian=TRUE was set

Author(s)

Jeff Laak

References

Ford, J. H., M. V. Bravington, and J. Robbins. 2012. Incorporating individual variability into mark-recapture models. Methods in Ecology and Evolution 3:1047-1054.

mstrata *Multistrata example data*

Description

An example data set which appears to be simulated data that accompanies MARK as an example analysis using the Multistrata model.

Format

A data frame with 255 observations on the following 2 variables.

ch a character vector containing the encounter history of each bird with strata

freq the number of birds with that capture history

Details

This is a data set that accompanies program MARK as an example for the Multistrata model and is also in the RMark pacakge. Here I use it to show the 3 ways models can be fitted to multistrata data. The model MSCJS is not run because it requires ADMB or the exe constructed from ADMB which is not available if downloaded from CRAN.

Examples

```
data(mstrata)
ms1=process.data(mstrata,model="MSCJS",strata.labels=c("A","B","C"))
ms2=process.data(mstrata,model="hmmMSCJS",strata.labels=c("A","B","C"))
# strata.labels for MVMS models must be specified as a list because more than one variable
# can be used
ms3=process.data(mstrata,model="MVMSCJS",strata.labels=list(state=c("A","B","C")))
ms1.ddl=make.design.data(ms1)
ms2.ddl=make.design.data(ms2)
ms3.ddl=make.design.data(ms3)

# following requires ADMB or the exe constructed from ADMB and links set for ADMB
mod1=try(crm(ms1,ms1.ddl,model.parameters=list(Psi=list(formula=~-1+stratum:tostratum),
      p=list(formula=~time)),hessian=TRUE))
if(class(mod1)[1]!="try-error") mod1

mod2=crm(ms2,ms2.ddl,model.parameters=list(Psi=list(formula=~-1+stratum:tostratum),
      p=list(formula=~time)),hessian=TRUE)
mod2

mod3=crm(ms3,ms3.ddl,model.parameters=list(Psi=list(formula=~-1+stratum:tostratum),
      p=list(formula=~time)),hessian=TRUE)
mod3
```

`mvmscjs` *Fitting function for Multivariate Multistate CJS with uncertainty models*

Description

A function for computing MLEs for MVMS models following Johnson et al (2015) via ADMB. It works very much like mscjs but with more parameters. While this function is for fitting the model via ADMB, the documentation contained here is also relevant when this type of model is fitted with optimx and FORTRAN code (use.admb=F).

It is easiest to call this function or any model built in `marked` through the function `crm`. This function has been exported but to fit a model it should be called through the crm function which does some testing of the arguments to avoid errors and sets up the calling arguments appropriately.

Usage

```
mvmscjs(x, ddl, dml, model_data = NULL, parameters, accumulate = TRUE,
  initial = NULL, method, hessian = FALSE, debug = FALSE,
  chunk_size = 1e+07, refit, itnmax = NULL, control = NULL, scale,
  re = FALSE, compile = FALSE, extra.args = "", clean = TRUE, sup, ...)
```

Arguments

`x`	processed dataframe created by process.data
`ddl`	list of dataframes for design data; created by call to `make.design.data`
`dml`	list of design matrices created by `create.dm` from formula and design data
`model_data`	a list of all the relevant data for fitting the model including imat, Phi.dm,p.dm,Psi.dm,Phi.fixed,p.fixed,Psi. and time.intervals. It is used to save values and avoid accumulation again if the model was re-rerun with an additional call to cjs when using autoscale or re-starting with initial values. It is stored with returned model object.
`parameters`	equivalent to `model.parameters` in `crm`
`accumulate`	if TRUE will accumulate capture histories with common value and with a common design matrix for S and p to speed up execution
`initial`	list of initial values for parameters if desired; if each is a named vector from previous run it will match to columns with same name
`method`	method to use for optimization; see `optim`
`hessian`	if TRUE will compute and return the hessian
`debug`	if TRUE will print out information for each iteration
`chunk_size`	specifies amount of memory to use in accumulating capture histories; amount used is 8*chunk_size/1e6 MB (default 80MB)
`refit`	non-zero entry to refit
`itnmax`	maximum number of iterations
`control`	control string for optimization functions

scale	vector of scale values for parameters
re	if TRUE creates random effect model admbcjsre.tpl and runs admb optimizer
compile	if TRUE forces re-compilation of tpl file
extra.args	optional character string that is passed to admb if use.admb==TRUE
clean	if TRUE, deletes the tpl and executable files for amdb if use.admb=T
sup	supplemental index values for constructing mvms model
...	not currently used

Details

The mvmscjs model is quite flexible but also quite complex. Prior to using this model, read Johnson et al. (2015) and particularly Supplement B which goes through the code used in the analysis of Johnson et al. (2015). Supplement B is useful but there have been a number of changes to the code since the paper was written and it is far from complete. The documentation here will fill in some of those blanks.

There are 5 classes of parameters in an mvms model. They are 1)Phi - survival probability, 2) p-capture probability, 3) Psi - state transition probabilities, 4) delta - certainty/uncertainty state probabilities, 5) pi - initial state probability. The final class pi was described in Johnson et al. (2015) but was not implemented in marked at the time the paper was written. However, with the sealions example, the initial state is known for all releases and pi is not relevant.

Before I jump into a description of each parameter, I'll describe some characteristics that are common to all of the parameters. First, all of the parameters are probabilities and thus must be in the unit interval. To accomplish this each of the parameters uses either a logit or mlogit (multinomial logit) link function.

A logit link is log(theta/(1-theta))= a+bx... where theta is the probability (called a real parameter in MARK terminology) and a+bx... is the set of covariates (e.g. x) and working parameters (a,b) that are unbounded. Working parameters are called beta parameters in MARK terminology. The inverse logit link is 1/(1+exp(-a-bx)) for this simple example and it can be computed from the plogis function in R (plogis(a+bx)). A largish negative value of the link (eg -5) mean theta will be nearly 0 and and a largish positive value (+5) means theta is nearly 1.

The mlogit link is simply an extension of the logit link for more than 2 classes. For example, the logit link for survival is for the 2 classes alive/dead. I'll use the logit link as a simple example to describe the way mlogit links are formed in marked. If survival is Phi=1/(1+exp(-a))=exp(a)/(1+exp(a)) then mortality is 1-Phi = 1-exp(a)/(1+exp(a))= 1/(1+exp(a)). Imagine that we have 2 link values a - for survival and b - for mortality. Then we can rewrite the inverse logit function for Phi as exp(a)/(exp(b)+exp(a)) and then 1-Phi = exp(b)/(exp(b)+exp(a)). It is obvious that Phi+1-Phi=1 which must be the case. Now, if you actually tried to fit a model with both a and b the parameters a and b are not uniquely identifiable because there is only one parameter Phi. One of the values must be a reference cell which means the value is fixed. Typically we assume b=0 which yields exp(0)=1 and we get the logit link.

Now consider 3 classes (A,B,C) and we'll use the link values a,b,c for a generic parameter theta. As with the 2 class example, the probabilities are theta(A)=exp(a)/(exp(a)+exp(b)+exp(c)), theta(B)=exp(b)/(exp(a)+exp(b)+exp(
and theta(C)=exp(c)/(exp(a)+exp(b)+exp(c)). As before theta(A)+theta(B)+theta(C)=1 but for a multinomial with 3 classes there are only 2 free estimable parameters due to the constraint that they have to sum to 1. Just as with the example above, a reference cell must be selected. If we chose

B as the reference cell then b=0 and the probabilities become theta(A)=exp(a)/(exp(a)+1+exp(c)), theta(B)=1/(exp(a)+1+exp(c)) and theta(C)=exp(c)/(exp(a)+1+exp(c)). This extends to an k-class multinomial where k-1 parameters are freely estimable and 1 must be fixed.

For parameters with logit links there is an obvious choice for the reference cell. For example, for survival probability we are interested in describing the parameters that affect survival, so we use death as the reference cell which is computed by subtraction (1-Phi). Likewise, for capture (sighting) probability we use not caught (1-p) as the reference cell. For mlogit links, there is not always an obvious choice for a reference cell and it can be an advantageous to allow the reference cell to be flexible and user-defined.

To understand how to set the reference cell, you should review the material in the help file for create.dmdf and Supplement B of Johnson et al. (2015) which describes the design data but I'll give a brief description here. All model building in the `marked` package revolves around the design data. If the design data are incorrect the model will be incorrect. Briefly, the design data contain the data for the parameter model, providing the maximum structure from which specific models can be built.. For example, for p there are M-1 records for each of the I individuals and M occasions (note that if the data are accumulated then an "individual" is actually a unique set of capture history and covariate data that represents "freq" animals). Thus, capture probability could be allowed to differ for each occasion, individual, age, strata, or any combination thereof. In practice, such a model would be overparameterized, but this structure allows maximum flexibility; users still need to use formula to provide specific constraints to reduce the number of parameters to a reasonable level. With this design data approach essentially there is a place holder for each possible p and any or all of those values can be fixed to a real value.

"Real" parameters can be fixed by adding a field called fix to the design data for a parameter. The value of fix should be either NA if the parameter is to be estimated or it should be the value of the real parameter (inverse link value). For example, we could fix the capture probability for time 2 to be 0 for all individuals at time 2 or at time 2 in strata B because there was no capture effort. Or you could fix p=0 for an individual at a time because it was temporarily removed. Or you could fix p=1 for a set of individuals that were always recaptured via telemetry.

Now let's go back to the mlogit parameters. There is an important difference between the way `marked` and `RMark/MARK` work with regard to mlogit parameters like Psi for state transition probabilities. An mlogit parameter is one in which the sum of the probabilities is 1. For Psi, if I'm in stratum A and can go to B or C or remain in A, the probabilities A to A, A to B and A to C must sum to 1 because that is all of the possibilities. In `RMark/MARK` the design data would only contain 2 records which are determined based on what you select as subtract.stratum. If subtract.stratum was set as A for the A stratum, the design data for Psi would only contain records for A to B and A to C. The value for A to A would be computed by subtraction and is the reference cell.

By contrast, in `marked`, all 3 records are in the design data and to set the transition from A to A as the reference cell (computed by subtraction) the value of fix is set to 1. So if fix is the inverse logit value why is it being set to 1 for the mlogit? Because the link for these parameters is actually a log-link and the inverse link is exp(link). So the link value is fixed to 0 and the inverse link ("real value") is exp(0)=1. The mlogit parameter is constructed by dividing each exp() value by the sum of the exp() values across the set defining the possible classes for the multinomial. Following along with the example,

```
Psi_AtoB=exp(beta_AtoB)/(1+exp(beta_AtoB)+exp(beta_AtoC))
Psi_AtoC=exp(beta_AtoC)/(1+exp(beta_AtoB)+exp(beta_AtoC)) and
Psi_AtoA=1-Psi_AtoB-Psi_AtoC = 1/(1+exp(beta_AtoB)+exp(beta_AtoC)).
```

The "appropriate" set will depend on the mlogit parameter and the data/model structure.

I used this structure for several reasons. First, you get a real parameter estimate for the subtracted stratum which you don't get in RMark/MARK. Secondly, you can change the value to be subtracted at will and it is not fixed across the entire model fit, but you do have to be careful when specifying the model when you do that because the formula specifies the parameters for those that are not fixed. Finally, you can change the reference cell simply by modifying the value of fix, whereas with the subtract.stratum approach the design data would have to be recreated which can take some time with large data sets.

Next I'll briefly describe each parameter in the model and its link function structure.

Phi —— Phi is survival probability in the model. Survival probability is also called S in the straight multi-state model. In retrospect I wish I had stayed with S but I didn't. In capture-recapture Phi is usually reserved for apparent survival rate which might include permanent emigration. With a multi-state model, the presumption is that "all" states are covered and animals can't emigrate permanently to some unobservable state. Whether that is true or not is always a question. Anyhow I use Phi for survival in mvmscjs models.

Phi uses a logit link (log(Phi/(1-Phi))= a+bx...) to relate survival to any covariates which have parameter values across the entire real line. Phi is what I call an interval parameter in that it describes the probability of surviving during an interval of time between sampling occasions. The design data for interval parameters are describe in terms of the time and age of the animal at the beginning of the interval. For example, if occasions are at time 1,2,...J there will be J-1 data records for each individual with time values 1,...,J-1.

p —— p is recapture/resighting probability. While I'll often fall back and call it capture or sighting probability, technically it is recapture or resighting because this model is based on a Cormack-Jolly-Seber (CJS) formulation which does not model initial "capture" probability and only models events after they are marked and released. p also uses the logit link and it is an occasion parameter because it describes an event associated with an occasion. Because they are "recaptures", there are J-1 design data records for each individual at times 2,...,J.

An example of using fix for p is given in the sealions example. For the data, there was no survey effort in stratum A for the last occasion.

```
ddl$p$fix = ifelse(ddl$p$Time==17 & ddl$p$area=="A", 0, ddl$p$fix)
```

Psi —— Psi describes the transition probabilities of moving between states. For each individual at each occasion for each of the possible states, it is formulated as a multinomial with a cell probability for movement from the state to all of the other possible states and remaining in the in the same state. Thus if there are M states, then there are M*M records for each individual for each of the J-1 intervals. It is treated as interval parameter with movement occurring between occasions but after survival. If an animal is in state A it survives and moves to state B with probability Phi_A*Psi_AtoB. If those were to be reversed you could use a trick proposed by Bill Kendall of adding dummy occasions between each real set of occasions and fixing real parameters. For example using the first 2 occasions with data A0, the data would become A00 with 3 occasions. For the first interval all the Phis would be fixed to 1 and Psi_AtoB would be estimated. For occasion 2, all the ps would be fixed to 0 because there are no observations and for the second interval all the Psi's should be fixed so it will remain in the same state (eg. Psi_AtoA=1 and Psi_Atox=0).

This can get rather involved when you have more than 1 variable describing the states which is the whole purpose of the multi-variate multi-state model. It just needs some careful thought. As an

example, here are some of the columns of the design data for the first interval for the first individual for stratum A– which for this example is area A with missing left and right tags. There are 8 records because there are 8 possible strata - 2 areas *2 left tag states * 2 right tag states.

```
> head(ddl$Psi[,c(1:10,20),],8)
 id occ stratum tostratum area ltag rtag toarea toltag tortag fix
  1   1     A--       A--    A    -    -      A      -      -   1
  1   1     A--       A-+    A    -    -      A      -      +   0
  1   1     A--       A+-    A    -    -      A      +      -   0
  1   1     A--       A++    A    -    -      A      +      +   0
  1   1     A--       S--    A    -    -      S      -      -  NA
  1   1     A--       S-+    A    -    -      S      -      +   0
  1   1     A--       S+-    A    -    -      S      +      -   0
  1   1     A--       S++    A    -    -      S      +      +   0
```

When the design data are created, it creates fields based on the definition of strata.labels in the call to process.data. For this example it was strata.labels=list(area=c("A","S"),ltag=c("+","-","u"),rtag=c("+","-","u"))). The design data include a stratum and tostratum field which are the concatenated values of all the variables defining the states. In addition field and tofield variables are created for each strata variables. Here the fields are area,ltag,rtag and the tofields are toarea,toltag and tortag. Each simply partitions the stratum and tostratum fields. Note that if only a single variable is used to define the strata.labels the the design data will only contain a stratum and tostratum fields.

Now, when the design data are initially created for Psi, it creates the field fix and assigns the value NA to all values except for the record in which stratum=tostratum which gets the value of fix=1 making it the reference cell. Those are the default values and it is up to you to change as needed. In this case almost all of the Psi values are fixed to 0 because they cannot occur. The only possible movement is from A– to S– because all of the other transitions would have it gaining tags that have already been lost. As another example, the stratum A++ would have all of the values of fix=NA except for tostratum=A++ which would be fixed to 1.

delta —– delta is the probability of being certain or uncertain about each state variable for a given stratum. The definition can change based on how you define the reference cell. For delta, the number of records depends on the structure of strata.labels and the number of state variables and how many state variables can be uncertain. Below are the design data for a single stratum for a single individual and occasion from the sealions example.

```
id occ stratum area ltag rtag obs.area obs.ltag obs.rtag obs.ltag.u obs.rtag.u
1   1     A--    A    -    -        A        -        -          0          0
1   1     A--    A    -    -        A        -        u          0          1
1   1     A--    A    -    -        A        u        -          1          0
1   1     A--    A    -    -        A        u        u          1          1
```

For this example, the area is certain but the left tag and right tag status can be uncertain. The above data are for the stratum A–. Each field used to define the states has a field value and an obs.field value (eg., ltag, obs.ltag). The fields obs.ltag.u and obs.rtag.u were created in code after the call to make.design.data (see ?sealions). There would be a set like these for each of the 8 possible stratum value for id=1 and occ=1. Because each tag status can be either known or uncertain there are 2*2 records and the sum of the 4 cell probabilities must sum to 1. The easiest way to do that is by

specifying fix=1 for one of the cells and assigning the other 3 fix=NA so they are estimated. If you set fix=1 for the first record (observation A–), then the parameters for delta are describing the probability of being uncertain. Whereas, if you set fix=1 for the fourth record (observation Auu) then the parameters for delta are describing the probability of being certain.

Now the use of fix is not required to set a reference cell because you can accomplish the same thing by defining the formula for delta such that the link value is 0 for one of the cells. For example, the formula for delta in the example is

```
~ -1 + obs.ltag.u + obs.rtag.u + obs.ltag.u:obs.rtag.u)
```

The numeric value of obs.ltag.u + obs.rtag.u + obs.ltag.u*obs.rtag.u (: equivalent to mulitplication here) is 0 for the first record and -1 in the formula specifies that there shouldn't be an intercept. Thus regardless of the 3 beta values for delta, the link value for the first record is 0 and the inverse link value is exp(0)=1 which is equivalent to setting fix=1 for the first record to set it as the reference cell. If you run the sealions example with the current version of the software it will give the message:

```
No values provided for fix for delta. At least need to set a reference cell
```

Messages like this are warnings and do not stop the model from being fitted because as shown above it is possible to set the reference cell with an appropriate formula. If a reference cell is not set the beta estimates are not all identifiable. Symptoms include troubles with optimization, different starting values giving different beta estimates for the parameter with the same likelihood value, and most easily seen is large standard errors for all of the beta estimates involved.

pi —— pi is the probability of being in a state for the initial occasion. It is only used when there is one or more individual capture histories have an uncertain (u) state variable value for the initial occasion of the capture history. For the sealions example, all are released in the state S++ so there was no uncertainty.

The design data for pi includes a record for each possible stratum for each individual. For the sealions example, the following are the design data for the first individual:

id	occ	stratum	area	ltag	rtag	fix
1	1	A--	A	-	-	0
1	1	A-+	A	-	+	0
1	1	A+-	A	+	-	0
1	1	A++	A	+	+	0
1	1	S--	S	-	-	0
1	1	S-+	S	-	+	0
1	1	S+-	S	+	-	0
1	1	S++	S	+	+	1

When the design data are created the code recognizes that the initial value for this individual capture history is known for all of the strata values so it assigns 0 to the strata that are not the initial value and 1 for the strata that is the initial value.

To demonstrate what is obtained when there is uncertainty I modified one of the S++ values to Suu and this is the resulting design data

```
id occ stratum area ltag rtag fix
335   1     A--    A    -    -   0
335   1     A-+    A    -    +   0
335   1     A+-    A    +    -   0
335   1     A++    A    +    +   0
335   1     S--    S    -    -   1
335   1     S-+    S    -    +  NA
335   1     S+-    S    +    -  NA
335   1     S++    S    +    +  NA
```

All of the fix values for area A are still 0 because area is certain and it is in area S. However, each tag status is uncertain so there are 4 possible true values. S– is set to the reference cell by default and the other 3 would be estimated.

There is a subtlety to the definition of pi that should be understood. The structure of the default design data for pi means that pi is the conditional probability of being in a state given there is uncertainty and that an animal is first observed in that period. In other words, it is the probability of being in states just for the records that have uncertainty and are first observed at a given time. For the sealions, consider having released 1/4 of the animals in A with all certain tag status and for the 3/4 released in S neither tag status was known. In that case pi would only relate to S and the proportion given both tags will possibly be less in S than in both areas combined. However, if you defined fix for all records as above the estimated pi would apply to the whole set.

The mvmscjs model is constructed as a hidden Markov model as described in Johnson et al. (2015). As such the 5 parameter classes (Phi,p,Psi,delta and pi) are used to construct the matrices gamma (eq 2.13 in Johnson et al. (2015)), dmat (eq 2.14 in Johnson et al. (2015)) and delta which is called pi in Johnson et al. (2015). In the code I used the name delta for the matrix which has I rows and M colums because the symbol delta is used in Zucchini et al's book on HMMs. So think of delta as the matrix and pi as the values in each row of the matrix. I should have used pi as the matrix as well to avoid confusion between the delta parameters and the matrix delta for initial values.

One other difference between the coding and Johnson et al. (2015) is that the 0 observation in dmat is the last row in Jonhson et al (2015) and it is the first row in the code. Likewise, for gamma the death state is the last row in Johnson et al. (2015) and the first row in the code. In Johnson et al. (2015) dmat and gamma are matrices but in the code they are 4-d arrays where the first 2 dimensions are individuals and occasions and the last 2 dimensions are the matrices defined in Johnson et al. (2015).

Johnson et al. (2015) show delta being used for each occasion including the initial release occasion. In fact, design data for delta are created for each occasion. But in constructing the formula there is one exception that you need to understand. If the initial values are either all certain or all uncertain then formula for delta is not used for that initial release because all values of dmat are either 0 or 1.

If these matrices are setup incorrectly the optimization will likely fail. In general, that is all handled by the code but it is possible to mess up the values in those matrices if the values of fix are incorrectly defined. One place where this can occur is if the value of fix=0 for all records defining a multinomial because when it normalizes to sum to 1, it will produce a NaN due to 0/0. To prevent problems the function HMMLikelihood with the argument check=TRUE is run before attempting the optimization. With the starting parameter values it checks to make sure that the columns of dmat sum to 1 and the rows of gamma sum to 1 and all values of pi in delta sum to 1. If any of these fail outside of rounding error, the program will issue errors showing the problematic records and values and will stop.

Unobservable states can be included in the data structure and model but you must be careful in constructing formulas. Keep in mind that there are no observations for unobservable states. That states the obvious but it can be easily forgotten when constructing formulas. For an easy example, consider a univariate example with area defining states. Will assume that we have areas A,B and C that are observable and X which is unobservable. For Psi you can have transitions to and from the X and the other states. Obviously, you'll need to set fix=0 for p for X because it can't be seen. Also, if you were to set the formula for Phi to be ~stratum there are no data in X so it will not be able to estimate survival in X. So don't do that. Likewise, the unobservable state value X should not be in a formula for delta because it represents certainty/uncertainty of state variables for observations and there are no observations in X. Also, because this is a CJS formulation that is conditional on the first release (observation), the formula for pi cannot include X, again because there are no observations.

To avoid including parameters for levels of a factor for unobservable state values, the easiest approach is to set fix values for the design data rows for the unobservable states. For example, if fix=0 for p for stratum X, then ~stratum could be used in the formula and it would have an intercept and parameters for B and C. The stratum level X would be excluded because all rows of the resulting design matrix with fixed values are set to 0 and then any columns which have all 0 values are dropped which would drop the X factor level. Another approach is to define a different variable that excludes the factor level. Note that this approach would not work to set p=0 for the unobservable state. Assume in addition to area we had another variable to define states and it could be uncertain. If we thought that the uncertainty in that variable was related to area, we could use a formula for delta like ~-1+notX:area where notX was a variable with value 1 when the area was not X and was 0 when area was X. This would result in parameters for A,B and C but not X. A similar approach would be to create 0/1 dummy variables A,B,C which were 1 when the record was for area=A,B,C respectively and 0 otherwise. Then you could use the delta formula ~-1+A+B+C specifically excluding X. It is not important for X to have a value for delta because there are no observations for X and if you look at the last row (observation =0) in eq 2.14 in Johnson et al. (2015) you'll see is 1-p and doesn't use delta.

`marked` doesn't include any specific robust design models because they can be relatively easily handled by fixing real parameters. In a standard closed multi-state robust design, during the secondary periods Phi=1 and Psi_AtoA= 1 for all states (no transitions). Any variation can be created simply by fixing the appropriate parameters. For example, with the sealions example, you might allow transitions between areas but assume no transitions for tag status.

Value

The resulting value of the function is a list with the class of crm,cjs such that the generic functions print and coef can be used.

`beta`	named vector of parameter estimates
`lnl`	-2*log likelihood
`AIC`	lnl + 2* number of parameters
`convergence`	result from `optim`; if 0 `optim` thinks it converged
`count`	`optim` results of number of function evaluations
`reals`	dataframe of data and real S and p estimates for each animal-occasion excluding those that occurred before release
`vcv`	var-cov matrix of betas if hessian=TRUE was set

Author(s)

Jeff Laake

References

Johnson, D. S., J. L. Laake, S. R. Melin, and DeLong, R.L. 2015. Multivariate State Hidden Markov Models for Mark-Recapture Data. 31:233-244.

---

mvms_design_data *Multivariate Multistate (mvms) Design Data*

Description

Creates a dataframe with design data for MvMS model for a single occasion

Usage

```
mvms_design_data(df.states, df = NULL, transition = TRUE)
```

Arguments

df.states	dataframe of states created from set_mvms
df	is dataframe of observations; provided for parameter delta in which both state and observation are needed in design data
transition	if TRUE, creates design data for a state transition (from to); otherwise just state variables

Value

a dataframe to be used with design data for mvms model

Author(s)

Jeff Laake

Examples

```
x=set_mvms(list(location=c("A","B","C"),repro_status=c("N","P","u")))
mvms_design_data(x$df.states)
mvms_design_data(x$df.states,transition=FALSE)
```

mvms_dmat *HMM Observation Probability matrix functions*

Description

Functions that compute the probability matrix of the observations given the state for various models. Currently only CJS, MS models and MS models with state uncertainty are included.

Usage

```
mvms_dmat(pars, m, F, T, sup)
```

Arguments

pars	list of real parameter matrices (id by occasion) for each type of parameter
m	number of states
F	initial occasion vector
T	number of occasions
sup	list of supplemental information that may be needed by the function but only needs to be computed once

Value

4-d array of id and occasion-specific observation probability matrices - state-dependent distributions in Zucchini and MacDonald (2009)

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. 275p.

omega *Compute 1 to k-step transition proportions*

Description

Computes 1 to k-step forward transition proportions in each state with a single transition matrix or a 3-d array of transition matrices.

Usage

```
omega(x, k = NULL, labels = NULL)
```

Arguments

x	either a transition matrix, list of transition matrices or 3-d array (a set of transition matrices)
k	if x is a transition matrix, this is number of steps 1 to k
labels	labels for states except for last which is always dead and is added at end

Author(s)

Jeff Laake

Phi.mean *Various utility parameter summary functions*

Description

Several functions have been added to compute mean values and boxplots of values. Currently they have only been defined for Phi and p for cjs and they are not generic.

Usage

```
Phi.mean(x,age=0,time=NULL,age.bins=NULL,age.levels=NULL)

       p.mean(x,age=0,time=NULL,age.bins=NULL,age.levels=NULL)

       Phi.boxplot(x,age=0,time=NULL,sex=NULL)

       p.boxplot(x,age=0,time=NULL,sex=NULL)
```

Arguments

x	dataframe of reals contained in model
age	at which Phi or p should be shown across time
time	at which Phi or p should be shown across ages
age.bins	bins for age in which values are summarized
age.levels	labels for age.bins
sex	for which Phi or p should be shown across ages

Value

matrix of labelled values for Phi.mean and p.mean or boxplot object

Author(s)

Jeff Laake

---

predict.crm *Compute estimates of real parameters*

Description

Computes real estimates and their var-cov for a particular subset of parameters.

Usage

```
## S3 method for class 'crm'
predict(object,newdata=NULL,ddl=NULL,parameter=NULL,unique=TRUE,
                   vcv=FALSE,se=FALSE,chat=1,subset,select,...)
```

Arguments

object	model object;
newdata	a dataframe for crm
ddl	list of dataframes for design data; created by call to make.design.data
parameter	name of real parameter to be computed (eg "Phi" or "p")
unique	TRUE if only unique values should be returned
vcv	logical; if TRUE, computes and returns v-c matrix of real estimates
se	logical; if TRUE, computes std errors and conf itervals of real estimates
chat	over-dispersion value
subset	logical expression using fields in real dataframe
select	character vector of field names in real that you want to include
...	generic arguments not used here

Value

A data frame (real) is returned if vcv=FALSE; otherwise, a list is returned also containing vcv.real:

real	data frame containing estimates, and if vcv=TRUE it also contains standard errors and confidence intervals
vcv.real	variance-covariance matrix of real estimates

Author(s)

Jeff Laake

Examples

```
data(dipper)
dipper.proc=process.data(dipper,model="cjs",begin.time=1)
dipper.ddl=make.design.data(dipper.proc)
mod.Phisex.pdot=crm(dipper.proc,dipper.ddl,
   model.parameters=list(Phi=list(formula=~sex+time),p=list(formula=~1)),hessian=TRUE)
xx=predict(mod.Phisex.pdot,ddl=dipper.ddl)
xx
xx=predict(mod.Phisex.pdot,newdata=dipper[c(1,23),],vcv=TRUE)
xx
```

---

print.crm *Print model results*

Description

Provides a printed simple summary of the model results.

Usage

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

Arguments

x	crm model result or list of model results
...	generic arguments not used here

Value

prints a simple summary of the model to the screen and returns NULL.

Author(s)

Jeff Laake

See Also

crm

`print.crmlist` *Print model table from model list*

Description

Print model table from model list

Usage

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

Arguments

`x`	list of model results
`...`	generic arguments not used here

Value

None

Author(s)

Jeff Laake

See Also

crm

`probitCJS` *Perform MCMC analysis of a CJS model*

Description

Takes design data list created with the function make.design.data for model "probitCJS" and draws a sample from the posterior distribution using a Gibbs sampler.

Usage

```
probitCJS(ddl, dml, parameters, design.parameters, burnin, iter,
  initial = NULL, imat = NULL)
```

Arguments

ddl	list of dataframes for design data; created by call to make.design.data
dml	list of design matrices created by create.dm from formula and design data
parameters	A model specification list with a list for Phi and p containing a formula and optionally a prior specification which is a named list. See 'Priors' section below.
design.parameters	Specification of any grouping variables for design data for each parameter
burnin	number of iteration to initially discard for MCMC burnin
iter	number of iteration to run the Gibbs sampler for following burnin
initial	A named list (Phi,p). If null and imat is not null, uses cjs.initial to create initial values; otherwise assigns 0
imat	A list of vectors and matrices constructed by process.ch from the capture history data

Value

A list with MCMC iterations and summarized output:

beta.mcmc	list with elements Phi and p which contain MCMC iterations for each beta parameter
real.mcmc	list with elements Phi and p which contain MCMC iterations for each real parameter
beta	list with elements Phi and p which contain summary of MCMC iterations for each beta parameter
reals	list with elements Phi and p which contain summary of MCMC iterations for each real parameter

Prior distribution specification

The prior distributions used in probitCJS are multivatiate normal with mean mu a and variance tau*(X'X)^-1 on the probit scale for fixed effects. The matrix X is the design matrix based on the model specification (located in parametersPhiformula and parameters$p$formula respectively). Priors for random effect variance components are inverse gamma with shape parameter 'a' and rate parameter 'b'. Currently, the default values are mu=0 and tau=0.01 for phi and p parameters. For all randome effects deault values are a=2 and b=1.0E-4. In addition to the variance component each random effect can be specified with a known unscaled covariance matrix, Q, if random effects are correlated. For example, to obtain a random walk model for a serially correlated effect see Examples section below. To specify different hyper-parameters for the prior distributions, it must be done in the parameters list. See the Examples section for changing the prior distributions. Note that a and b can be vectors and the Qs are specified via a list in order of the random effects specified in the model statements.

Author(s)

Devin Johnson

Examples

```
# This example is excluded from testing to reduce package check time
# Analysis of the female dipper data
data(dipper)
dipper=dipper[dipper$sex=="Female",]
# following example uses unrealistically low values for burnin and
# iteration to reduce package testing time
fit1 = crm(dipper,model="probitCJS",model.parameters=list(Phi=list(formula=~time),
 p=list(formula=~time)), burnin=100, iter=1000)
fit1
# Real parameter summary
fit1$results$reals

# Changing prior distributions:
fit2 = crm(dipper,model="probitCJS",
   model.parameters=list(
     Phi=list(formula=~time, prior=list(mu=rep(0,6), tau=1/2.85^2)),
     p=list(formula=~time, prior=list(mu=rep(0,6), tau=1/2.85^2))
   ), burnin=100, iter=1000)
fit2
# Real parameter summary
fit2$results$reals

# Creating a Q matrix for random walk effect for 6 recapture occasions
A=1.0*(as.matrix(dist(1:6))==1)
Q = diag(apply(A,1,sum))-A

# Fit a RW survival process
fit3 = crm(dipper,model="probitCJS",
   model.parameters=list(
     Phi=list(
       formula=~(1|time),
       prior=list(mu=0, tau=1/2.85^2, re=list(a=2, b=1.0E-4, Q=list(Q)))
       ),
     p=list(formula=~time, prior=list(mu=rep(0,6), tau=1/2.85^2))
   ), burnin=100, iter=1000)
fit3
# Real parameter summary (Not calculated for random effects yet)
fit3$results$reals
```

`proc.form` *Mixed effect model formula parser Parses a mixed effect model in the lme4 structure of ~fixed +(re1|g1) +...+(ren|gn)*

Description

Mixed effect model formula parser

Parses a mixed effect model in the lme4 structure of ~fixed +(re1|g1) +...+(ren|gn)

Usage

```
proc.form(f)
```

Arguments

f	formula for mixed effect mode in the form used in lme4; ~fixed +(re1|g1) +...+(ren|gn)

Value

A list with elements fix.model and re.model. fix.model contains the formula for the fixed effects; re.model contains elements sub, the grouping formula and model the design formula for the random effect. Each formula is of type character and must be wrapped with as.formula in use with model.matrix

Author(s)

Devin Johnson <devin.johnson@noaa.gov>

---

process.ch *Process release-recapture history data*

Description

Creates needed constructs from the release-recapture history.

Usage

```
process.ch(ch, freq = NULL, all = FALSE)
```

Arguments

ch	Vector of character strings; each character string is composed of either a constant length sequence of single characters (01001) or the character string can be comma separated if more than a single character is used (1S,1W,0,2W). If comma separated, each string must contain a constant number of elements.
freq	Optional vector of frequencies for ch; if missing assumed to be a; if <0 indicates a loss on capture
all	FALSE is okay for cjs unless R code used to compute lnl instead of FORTRAN; must be true for js because it returns additional quantities needed for entry prob.

Value

nocc	number of capture occasions
freq	absolute value of frequency for each ch
first	vector of occasion numbers for first 1
last	vector of occasion numbers for last 1
loc	vector of indicators of a loss on capture if set to 1
chmat	capture history matrix
FtoL	1's from first (1) to last (1) and 0's elsewhere; only if all==TRUE
Fplus	1's from occasion after first (1) to nocc(last occasion); only if all==TRUE
Lplus	1's from occasion after last (1) to nocc; only if all==TRUE
L	1's from last (1) to nocc; only if all==TRUE

Author(s)

Jeff Laake

---

process.data *Process encounter history dataframe for MARK analysis*

Description

Prior to analyzing the data, this function initializes several variables (e.g., number of capture occasions, time intervals) that are often specific to the capture-recapture model being fitted to the data. It also is used to 1) define groups in the data that represent different levels of one or morestrata.labels factor covariates (e.g., sex), 2) define time intervals between capture occasions (if not 1), and 3) create an age structure for the data, if any.

Usage

```
process.data(data,begin.time=1,model="CJS",mixtures=1,groups=NULL,
                       allgroups=FALSE,age.var=NULL,initial.ages=c(0),
                       time.intervals=NULL,nocc=NULL,accumulate=TRUE,
                       strata.labels=NULL)

         accumulate_data(data)
```

Arguments

data	A data frame with at least one field named ch which is the capture (encounter) history stored as a character string. data can also have a field freq which is the number of animals with that capture history. The default structure is freq=1 and it need not be included in the dataframe. data can also contain an arbitrary number of covariates specific to animals with that capture history.
begin.time	Time of first capture occasion or vector of times if different for each group

model	Type of analysis model.
mixtures	Number of mixtures in closed capture models with heterogeneity
groups	Vector of factor variable names (in double quotes) in data that will be used to create groups in the data. A group is created for each unique combination of the levels of the factor variables in the list.
allgroups	Logical variable; if TRUE, all groups are created from factors defined in groups even if there are no observations in the group
age.var	An index in vector groups for a variable (if any) for age
initial.ages	A vector of initial ages that contains a value for each level of the age variable groups[age.var]; if the data contain an initial.age field then it will be used instead.
time.intervals	Vector of lengths of time between capture occasions or matrix of time intervals with a row for each animal and column for each interval between occasions.
nocc	number of occasions for Nest type; either nocc or time.intervals must be specified
accumulate	if TRUE, aggregates data with same values and creates freq field for count of records
strata.labels	labels for strata used in capture history; they are converted to numeric in the order listed. Only needed to specify unobserved strata; for any unobserved strata p=0.

Details

For examples of data, see dipper. The structure of the encounter history and the analysis depends on the analysis model to some extent. Thus, it is necessary to process a dataframe with the encounter history (ch) and a chosen model to define the relevant values. For example, number of capture occasions (nocc) is automatically computed based on the length of the encounter history (ch) in data. Currently, only 2 types of models are accepted in marked: cjs and js. The default time interval is unit time (1) and if this is adequate, the function will assign the appropriate length. A processed data frame can only be analyzed using the model that was specified. The model value is used by the functions make.design.data and crm to define the model structure as it relates to the data. Thus, if the data are going to be analysed with different underlying models, create different processed data sets with the model name as an extension. For example, dipper.cjs=process.data(dipper).

This function will report inconsistencies in the lengths of the capture history values and when invalid entries are given in the capture history.

The argument begin.time specifies the time for the first capture occasion and not the first time the particular animal was caught or releaed. This is used in creating the levels of the time factor variable in the design data and for labelling parameters. If the begin.time varies by group, enter a vector of times with one for each group. It will add a field begin.time to the data with the value for each individual. You can also specify a begin.time field in the data allowing each animal to have a unique begin.time. Note that the time values for survivals are based on the beginning of the survival interval and capture probabilities are labeled based on the time of the capture occasion. Likewise, age labels for survival are the ages at the beginning times of the intervals and for capture probabilities it is the age at the time of capture/recapture.

The time.intervals argument can either be a vector of lengths of times for each interval between occasions that is constant for all animals or a matrix which has a row for each animal and a column

for each interval which lets the intervals vary by animals. These intervals are used to construct the design data and are used for the field time.interval which is used to adjust parameters like Phi and S to a constant per unit time interval (eg annual survival rates). On occasion it can be useful to leave the time.interval to remain at default of 1 or some other vector of time.intervals to construct the design data and then modify the time.interval value in the design data. For example, assume that cohort marking and release is done between sampling occasions. The initial survival from release to the next sampling occasion may vary by release cohort, but the remainder of the surivivals are between sampling occasions. In that case it is easier to let time.interval=1 (assuming unit interval (eg year) between sampling occasions but then modifying ddlPhitime.interval to the value for the first interval after each release to be the partial year from release to next sampling occasion. In this way everything is labelled with annual quantities but the first partial year survival is adjusted to an annual rate.

Note that if you specify time.intervals as a matrix, then accumulate is set to FALSE so that the number of rows in the data can be checked against the number of rows in the time.intervals matrix and thus data cannot be accumulated because at present it doesn't use values of time.intervals to determine which records can be accumulated.

`groups` is a vector of variable names that are contained in `data`. Each must be a factor variable. A group is created for each unique combination of the levels of the factor variables. In the first example given below `groups=c("sex","age","region")`. which creates groups defined by the levels of `sex`, `age` and `region`. There should be 2(sexes)*3(ages)*4(regions)=24 groups but in actuality there are only 16 in the data because there are only 2 age groups for each sex. Age group 1 and 2 for M and age groups 2 and 3 for F. This was done to demonstrate that the code will only use groups that have 1 or more capture histories unless `allgroups=TRUE`.

The argument `age.var=2` specifies that the second grouping variable in `groups` represents an age variable. It could have been named something different than age. If a variable in `groups` is named `age` then it is not necessary to specify `age.var`. `initial.age` specifies that the age at first capture of the age levels. For example initial.age=0:2 specifies that the initial.ages are 0,1 and 2 for the age class levels designated as 1,2,3. The actual ages for the age classes do not have to be sequential or ordered, but ordering will cause less confusion. Thus levels 1,2,3 could represent initial ages of 0,4,6 or 6,0,4. The default for `initial.age` is 0 for each group, in which case, `age` represents time since marking (first capture) rather than the actual age of the animal. If the data contains an initial.age field then it overrides any other values and lets each animal have a unique initial.age at first capture/release.

The following variable names are reserved and should be used as follows: id (animal id) ch(capture history) freq (number of animals with that ch/data) The following variable names are reserved and should not be used in the data: occ,age,time,cohort,Age,Time,Cohort,Y,Z,initial.age,begin.time,time.interval,fix

Value

from `process.data` processed.data (a list with the following elements)

`data`	original raw dataframe with group factor variable added if groups were defined
`model`	type of analysis model (eg, "cjs" or "js")
`freq`	a dataframe of frequencies (same # of rows as data, number of columns is the number of groups in the data. The column names are the group labels representing the unique groups that have one or more capture histories.
`nocc`	number of capture occasions

time.intervals	length of time intervals between capture occasions
begin.time	time of first capture occasion
initial.ages	an initial age for each group in the data; Note that this is not the original argument but is a vector with the initial age for each group. In the first example below proc.example.data$initial.ages is a vector with 16 elements as follows 0 1 1 2 0 1 1 2 0 1 1 2 0 1 1 2
group.covariates	factor covariates used to define groups

from accumulate_data a dataframe with same column structure as argument with addition of freq (if not any) and reduced to unique rows with freq accumulating number of records.

Author(s)

Jeff Laake

See Also

dipper,crm

Examples

```
data(dipper)
dipper.process=process.data(dipper)
accumulate_data(dipper)
```

---

resight.matrix *Various utility functions*

Description

Several functions have been added to help visualize data including resight.matrix which provides for each cohort the number of releases on the diagonal and the number resighted on each occasion in the upper-triangular matrix. naive.survival provides a naive survival estimate at each time for each cohort. The estimate for time i is the number resighted at time i+1 or later divided by the number seen at time i or later. These values should be interpreted cautiously because they are influenced by capture probability in year i but it is useful to identify particularly high or low survival values. Functions Phi.mean and p.mean compute average real parameter values Phi or p across time for a single age or across ages for a single time.

Usage

```
resight.matrix(x)

       naive.survival(x)

       mcmc_mode(x)
```

Arguments

x	processed data list - result from process.data in marked or real estimates from fitted model

Value

matrix of values with cohort and year labels

Author(s)

Jeff Laake

R_HMMLikelihood *Hidden Markov Model Functions*

Description

R implementation of HMMs described in processed report except function HMMLikelihood renamed to R_HMMLikelihood and changed to compute values for all capture histories and return lnl, alpha, phi, v, dmat, and gamma values. loglikelihood is called with a fitted hmm model and then computes the gamma,dmat and delta matrices and calls R_HMMLikelihood function. These are not used by the fitting code.

Usage

```
R_HMMLikelihood(x,first,m,T,dmat,gamma,delta)
       loglikelihood(object,ddl=NULL)
```

Arguments

x	single observed sequence (capture history)
first	occasion to initiate likelihood calculation for sequence
m	number of states
T	number of occasions; sequence length
dmat	observation probability matrices
gamma	transition matrices
delta	initial distribution
object	fitted hmm model
ddl	design data list; will be computed if NULL

Value

both return log-likelihood, alpha, v and phi arrays

Author(s)

Jeff Laake

References

Zucchini, W. and I.L. MacDonald. 2009. Hidden Markov Models for Time Series: An Introduction using R. Chapman and Hall, Boca Raton, FL. 275p. See page 45.

`sealions` *Multivariate State example data*

Description

An example data set using California sea lions to demonstrate the Multivariate state model with 3 variables defining the states : area, ltag and rtag. The left tag (ltag) and right tag (rtag) variables can be unknown. #'

Format

A data frame with 485 observations on the following 3 variables.

ch a character vector of 3-character values for each occasion separated by commas.

sex the sex of the sea lion designated as F or M

weight the anomaly of the weight from the sex-specific mean

Examples

```
### Load packages ###
# The splines package is only necessary for fitting b-spline curves used in the paper
# It is not required for the multivate models in the marked package
library(splines)

# Get data
data(sealions)

# Process data for multivariate models in marked
dp=process.data(sealions,model="mvmscjs",
  strata.labels=list(area=c("A","S"),ltag=c("+","-","u"),rtag=c("+","-","u")))

### Make design data
ddl=make.design.data(dp)

# Create pup variable for Phi
ddl$Phi$pup=ifelse(ddl$Phi$Age==0,1,0)
ddl$Phi$sex=factor(ddl$Phi$sex)

# Detection model
```

```
# Set final year (2014)  p=0 (no resight data) for ANI
ddl$p$fix = ifelse(ddl$p$Time==17 & ddl$p$area=="A", 0, ddl$p$fix)

# Delta model
# create indicator variables for 'unknown' tag observations
ddl$delta$obs.ltag.u = ifelse(ddl$delta$obs.ltag=="u", 1, 0)
ddl$delta$obs.rtag.u = ifelse(ddl$delta$obs.rtag=="u", 1, 0)

# Psi model
# Set Psi to 0 for cases which are not possible - missing tag to having tag
ddl$Psi$fix[as.character(ddl$Psi$ltag)=="-"&as.character(ddl$Psi$toltag)=="+"]=0
ddl$Psi$fix[as.character(ddl$Psi$rtag)=="-"&as.character(ddl$Psi$tortag)=="+"]=0
# Create indicator variables for transitioning between states
ddl$Psi$AtoS=ifelse(ddl$Psi$area=="A"&ddl$Psi$toarea=="S",1,0)  # ANI to SMI movement
ddl$Psi$StoA=ifelse(ddl$Psi$area=="S"&ddl$Psi$toarea=="A",1,0)  # SMI to ANI movement
ddl$Psi$lpm=ifelse(ddl$Psi$ltag=="+"&ddl$Psi$toltag=="-",1,0)   # Losing left tag
ddl$Psi$rpm=ifelse(ddl$Psi$rtag=="+"&ddl$Psi$tortag=="-",1,0)   # Losing right tag
ddl$Psi$sex=factor(ddl$Psi$sex)

# formulas
Psi.1=list(formula=~-1+ AtoS:sex + AtoS:sex:bs(Age) + StoA:sex + StoA:sex:bs(Age) +
                     I(lpm+rpm) +I(lpm+rpm):Age + lpm:rpm)
p.1=list(formula=~time*area)
delta.1=list(formula= ~ -1 + obs.ltag.u + obs.rtag.u + obs.ltag.u:obs.rtag.u)
Phi.1=list(formula=~sex*bs(Age)+pup:weight+area)

# Fit model - commented out because it takes >1hr to run
# mod=crm(dp,ddl,model.parameters=list(Psi=Psi.1,p=p.1,delta=delta.1,Phi=Phi.1),hessian=TRUE)
```

`set.fixed` *Set fixed real parameter values in ddl*

Description

Merges fixed real parameter values in parameter specification with those specified with fix in the design data list and returns the fixed values in the design data list.

Usage

```
set.fixed(ddl, parameters)
```

Arguments

`ddl`	design data list
`parameters`	parameter specififcation list

Value

design data list with fixed values

Author(s)

Jeff Laake

`set.initial` *Set initial values*

Description

Sets initial values specified in a list.

Usage

```
set.initial(pars, dml, initial)
```

Arguments

`pars`	character vector of parameter names
`dml`	list of design matrices created by `create.dm` from formula and design data
`initial`	list of vectors for parameter initial values

Value

List of initial values for each parameter in the model

Author(s)

Jeff Laake

`set.scale` *Scaling functions*

Description

Set scale, scale dm and scale/unscale parameters

Usage

```
set.scale(pars,model_data,scale)

        scale.dm(model_data,scale)

         scale.par(par,scale)

         unscale.par(par,scale)
```

Arguments

pars	character vector of parameter names
model_data	list of data/design objects
scale	list or vector of parameter scales
par	list of parameter vectors or vector of parameter values

Value

List of scale values for set.scale, model.data with scaled design matrices for scale.dm, vector of scaled parameter values for scale.par, and list of unscaled parameter vectors for unscale.par

Author(s)

Jeff Laake

---

setup.model *Defines model specific parameters (internal use)*

Description

Compares model, the name of the type of model (eg "CJS") to the list of acceptable models to determine if it is supported and then creates some global fields specific to that type of model that are used to modify the operation of the code.

Usage

```
setup.model(model, nocc, mixtures = 1)
```

Arguments

model	name of model type (must be in vector valid.models)
nocc	length of capture history string
mixtures	number of mixtures

Details

In general, the structure of the different types of models (e.g., "CJS","Recovery",...etc) are very similar with some minor exceptions. This function is not intended to be called directly by the user but it is documented to enable other models to be added. This function is called by other functions to validate and setup model specific parameters. For example, for live/dead models, the length of the capture history is twice the number of capture occasions and the number of time intervals equals the number of capture occasions because the final interval is included with dead recoveries. Whereas, for recapture models, the length of the capture history is the number of capture occasions and the number of time intervals is 1 less than the number of occasions. This function validates that the model is valid and sets up some parameters specific to the model that are used in the code.

Value

model.list - a list with following elements

etype	encounter type string for MARK input; typically same as model
nocc	number of capture occasions
num	number of time intervals relative to number of occasions (0 or -1)
mixtures	number of mixtures if any
derived	logical; TRUE if model produces derived estimates

Author(s)

Jeff Laake

See Also

setup.parameters, valid.parameters

---

setup.parameters *Setup parameter structure specific to model (internal use)*

Description

Defines list of parameters used in the specified type of model (model) and adds default values for each parameter to the list of user specified values (eg formula, link etc).

Usage

```
setup.parameters(model, parameters = list(), nocc = NULL, check = FALSE,
  number.of.groups = 1)
```

Arguments

model	type of model ("CJS", "Burnham" etc)
parameters	list of model parameter specifications
nocc	number of occasions (value only specified if needed)
check	if TRUE only the vector of parameter names is returned par.list
number.of.groups	number of groups defined for data

Value

The return value depends on the argument check. If it is TRUE then the return value is a vector of the names of the parameters used in the specified type of model. For example, if model="CJS" then the return value is c("Phi","p"). This is used by the function valid.parameters to make sure that parameter specifications are valid for the model (i.e., specifying recovery rate r for "CJS" would give an error). If the function is called with the default of check=FALSE, the function returns a list of parameter specifications which is a modification of the argument parameters which adds parameters not specified and default values for all types of parameters that were not specified. The list length and names of the list elements depends on the type of model. Each element of the list is itself a list with varying numbers of elements which depend on the type of parameter although some elements are the same for all parameters. Below the return value list is shown generically with parameters named p1,...,pk.

p1	List of specifications for parameter 1
p2	List of specifications for parameter 2
.	
.	
.	
pk	List of specifications for parameter k

The elements for each parameter list all include:

begin	0 or 1; beginning time for the first parameter relative to first occasion
num	0 or -1; number of parameters relative to number of occassions
type	type of PIM structure; either "Triang" or "Square"
formula	formula for parameter model (e.g., ~time)
link	link function for parameter (e.g., "logit")

and may include:

share	only valid for p in closed capture models; if TRUE p and c models shared
mix	only valid for closed capture heterogeneity models; if TRUE mixtures are used
rows	only valid for closed capture heterogeneity models
fixed	fixed values specified by user and not used modified in this function

Author(s)

Jeff Laake

See Also

setup.model,valid.parameters

---

setup_admb *ADMB setup*

Description

Sets up executable for the tpl file by looking for exe in package directory or compiles tpl file in local directory (clean=F) of from package directory. If admb cannot be found will attempt to run prepare_admb function (if exists) to establish connections for compilation.

Usage

```
setup_admb(tpl, compile = FALSE, clean = TRUE, safe = TRUE, re = FALSE)
```

Arguments

tpl	character string for admb template file
compile	if TRUE forces re-compilation of tpl file
clean	if TRUE, deletes the tpl and executable files for amdb in local directory
safe	can be used to set safe mode for admb
re	uses admb-re if TRUE for random effects

---

setup_tmb *TMB setup*

Description

Sets up executable for the .cpp file (tpl) by looking for exe in package directory or compiles cpp file in local directory (clean=FALSE) of from package directory.

Usage

```
setup_tmb(tpl, clean = FALSE)
```

Arguments

tpl	character string for admb template file
clean	if TRUE, deletes the tpl (.cpp) and executable files in local directory and copies from package directory

`set_mvms` *Multivariate Multistate (mvms) Specification*

Description

Creates list data structure from mvms specification

Usage

```
set_mvms(x)
```

Arguments

`x`	a multivariate multistate (mvms) specification as described above

Details

Accepts a mvms specification which is a list with named character vectors and optionally a vector named exclude. The length of the list (except for exclude) is the multivariate dimension. The name of each dimension is the list element name. Each character vector specifies the one character labels for the states of that dimension and optionally a reserved character "u" to specify that there is state uncertainty for that dimension. The name vector exclude can be used to remove variable combinations that cannot occur in the data.

The code tests to make sure that the input mvms specification is of the correct structure and it stops with an error message if not. The code returns a list structure with a number of elements described under return value below.

Value

a list with the following elements: 1) mvms - the input specification, 2) nd - the number of dimensions, 3) df - the dataframe containing all combinations of observations across dimensions including uncertain states, 4) df.states - the dataframe with all combinations of states across dimensions, 5) uncertain - boolean vector with nd elements indicating whether there is uncertainy in states for each dimension.

Author(s)

Jeff Laake

Examples

```
set_mvms(list(location=c("A","B","C"),repro_status=c("N","P","u"),exclude=c("CP")))
```

simHMM *Simulates data from Hidden Markov Model*

Description

Creates a set of data from a specified HMM model for capture-recapture data.

Usage

```
simHMM(data, ddl = NULL, begin.time = 1, model = "hmmCJS", title = "",
  model.parameters = list(), design.parameters = list(), initial = NULL,
  groups = NULL, time.intervals = NULL, accumulate = TRUE,
  strata.labels = NULL)
```

Arguments

data	Either the raw data which is a dataframe with at least one column named ch (a character field containing the capture history) or a processed dataframe
ddl	Design data list which contains a list element for each parameter type; if NULL it is created
begin.time	Time of first capture(release) occasion
model	Type of c-r model
title	Optional title; not used at present
model.parameters	List of model parameter specifications
design.parameters	Specification of any grouping variables for design data for each parameter
initial	Optional list (by parameter type) of initial values for beta parameters (e.g., initial=list(Phi=0.3,p=-2)
groups	Vector of names of factor variables for creating groups
time.intervals	Intervals of time between the capture occasions
accumulate	if TRUE, like capture-histories are accumulated to reduce computation
strata.labels	labels for strata used in capture history; they are converted to numeric in the order listed. Only needed to specify unobserved strata. For any unobserved strata p=0..

Details

The specification for the simulation includes a set of data with at least 2 unique ch and freq value to specify the number of ch values to simulate that start at the specified occasion. For example, 1000 50 0100 50 0010 50 would simulate 150 capture histories with 50 starting at each of occasions 1 2 and 3. The data can also contain other fields used to generate the model probabilities and each row can have freq=1 to use individual covariates. Either a dataframe (data) is provided and it is processed and the design data list are created or the processed dataframe and design data list are provided. Formula for the model parameters for generating the data are provided in model.parameters and parameter values are provided in initial.

Value

dataframe with simulated data

Author(s)

Jeff Laake

Examples

```
# simulate phi(.) p(.) with 1000 Females and 100 males, 3 occasions all released on first occasion
df=simHMM(data.frame(ch=c("100","110"),sex=factor(c("F","M")),freq=c(1000,100),
   stringsAsFactors=FALSE))
df=simHMM(data.frame(ch=rep("100",100),u=rnorm(100,0,1),freq=rep(1,100),
  stringsAsFactors=FALSE),
  model.parameters=list(Phi=list(formula=~u),p=list(formula=~time)),
   initial=list(Phi=c(1,1),p=c(0,1)))
df=simHMM(data.frame(ch=c("1000","0100","0010"),freq=rep(50,3),stringsAsFactors=FALSE),
  model.parameters=list(Phi=list(formula=~1),p=list(formula=~time)),
    initial=list(Phi=c(1),p=c(0,1,2)))

########################################################################
# Example developed by Jay Rotella, Montana State University
# example of using the 'simHMM' function in 'marked' package for
#  a multi-state model with 3 states

# simulate a single release cohort of 1000 animals with 1 release from
#  each of the 3 states for 10 recapture occasions;
# Note: at least 2 unique ch are needed in simHMM
simd <- data.frame(ch = c("A0000000000", "B0000000000", "C0000000000"),
freq = c(1000, 1000, 1000),
stringsAsFactors = FALSE)

# define simulation/fitting model; default for non-specified parameters is ~1
# but all are listed here. For the formula for Psi, the -1 is necessary to remove
# the intercept which is not needed and would be redundant.
# The formula '~ -1 + stratum:tostratum' is often appropriate for multi-state
# transitions as it allows different values for Psi depending on
# the current state (state at time t) and the new state (state at t+1).
modelspec <- list(
S = list(formula = ~ -1 + stratum),
# by presenting the formula for S as '~-1+stratum', it is possible to
# present the desired survival rates directly for each group in the
# simulations (see how 'initial' is created below) for this simple
# scenario where survival varies by stratum but no other covariates
p = list(formula = ~ 1),
Psi = list(formula =  ~ -1 + stratum:tostratum))

# process data with A,B,C strata
sd <- process.data(simd,
model = "hmmMSCJS",
strata.labels = c("A", "B", "C"))
```

```
# create design data
ddl <- make.design.data(sd)
# view design data (especially important for seeing which order to present
#  beta values used to set probabilities for various transitions)
head(model.matrix(~stratum, ddl$S))
head(model.matrix(~-1+stratum:tostratum, ddl$Psi), 9)

# set initial parameter values for model S=~stratum, p=~1, Psi=~stratum:tostratum
initial <- list(S = c(log(0.8/0.2), log(0.6/0.4), log(0.5/0.5)),
p = log(0.6/0.4),
# order of presentation is BA, CA, AB, CB, AC, BC
# Note: values for AA, BB, CC are obtained by subtraction
Psi = c(log(0.2/0.6), log(0.3/0.1),   # Psi(BA) & Psi(CA)
log(0.3/0.5), log(0.6/0.1),   # Psi(AB) & Psi(CB)
log(0.2/0.5), log(0.2/0.6)))  # Psi(AC) & Psi(BC)
# The desired probabilties for the transition matrix for Psi are as follows
#        To:
# From:    A   B    C
#    A   .5   .3   .2
#    B   .2   .6   .2
#    C   .3   .6   .4
# The values in the list above are obtained using the transitions AA, BB, CC
#  as reference values in the denominator of log-odds calculations.
# For example, to achieve the desired probabilities for the transition
#  from A to B use log(0.3/0.5) and from A to C use log(0.2/0.5)
# exp(log(0.3/0.5))/(1 + exp(log(0.3/0.5)) + exp(log(0.2/0.5))) = 0.3
# exp(log(0.2/0.5))/(1 + exp(log(0.3/0.5)) + exp(log(0.2/0.5))) = 0.2
# Note:           1/(1 + exp(log(0.3/0.5)) + exp(log(0.2/0.5))) = 0.5

# call simmHMM to get a single realization
realization <- simHMM(sd, ddl, model.parameters = modelspec,
initial = initial)

# using that realization, process data and make design data
# note that the analysis can also use model="MSCJS" in marked or "Multistrata" in RMark
# it is only the simulation that requires specification as an HMM.
sd <- process.data(realization, model = "hmmMSCJS",
strata.labels = c("A","B","C"))
rddl <- make.design.data(sd)

# fit model
m <- crm(sd, rddl, model.parameters = modelspec,
hessian = TRUE)

# model output
m$results$beta
initial
m$results$reals
```

skagit *An example of the Mulstistrata (multi-state) model in which states are routes taken by migrating fish.*

Description

An example of the Mulstistrata (multi-state) model in which states are routes taken by migrating fish.

Format

A data frame with 100 observations on the following 2 variables.

ch capture history

tag tag type v7 or v9

Author(s)

Megan Moore <megan.moore at noaa.gov>

Examples

```
# There are just two states which correspond to route A and route B. There are 6 occasions
# which are the locations rather than times. After release at 1=A there is no movement
# between states for the first segment, fish are migrating downriver together and all pass 2A.
# Then after occasion 2, migrants go down the North Fork (3A) or the South Fork (3B),
# which both empty into Skagit Bay. Once in saltwater, they can go north to Deception Pass (4A)
# or South to a receiver array exiting South Skagit Bay (4B). Fish in route A can then only go
# to the Strait of Juan de Fuca, while fish in route B must pass by Admiralty Inlet (5B).
# Then both routes end with the array at the Strait of Juan de Fuca.
#
#       1A
#        |
#        2A
#      /    \
#    3A       3B
#   /  \     /  \
# 4A   4B   4A  4B
#  |    \   /   |
#   5A    5B  5A   5B
#      \   \   /    /
#            6
#
# from 3A and 3B they can branch to either 4A or 4B; branches merge at 6
# 5A does not exist so p=0; only survival from 4A to 6 can be
# estimated which is done by setting survival from 4A to 5A to 1 and
# estimating survival from 5A to 6 which is then total survival from 4A to 6.

# See help for mscjs for an example that explains difference between marked and RMark
# with regard to treatment of mlogit parameters like Psi.
```

splitCH *Split/collapse capture histories*

Description

splitCH will split a character string vector of capture histories into a matrix. The ch can either be single character or comma separated string. The matrix is appended to the original data set (data) if one is specified. Will handle character and numeric values in ch. Results will differ depending on content of ch. collapseCH will collapse a capture history matrix back into a character vector. Argument can either be a capture history matrix (chmat) or a dataframe (data) that contains fields with a specified prefix.

Usage

```
splitCH(x="ch", data=NULL, prefix="Time")

        collapseCH(chmat=NULL, data=NULL, prefix="Time", collapse="")
```

Arguments

x	A vector containing the character strings of capture histories or the column number or name in the data set data
data	A data frame containing columnwith value in x if x indicates a column in a data frame
prefix	first portion of field names for split ch
chmat	capture history matrix
collapse	in collapseCH the separator for ch string; defaults to "" but "," also useful if multi-characters are used

Value

A data frame if data specified and a matrix if vector ch is specified

Author(s)

Devin Johnson; Jeff Laake

Examples

```
data(dipper)
# following returns a matrix
chmat=splitCH(dipper$ch)
# following returns the original dataframe with the ch split into columns
newdipper=splitCH(data=dipper)
# following collapses chmat
ch=collapseCH(chmat)
# following finds fields in newdipper and creates ch
```

```
newdipper$ch=NULL
newdipper=collapseCH(data=newdipper)
```

---

tagloss *Tag loss example*

Description

A simulated data set of double tag loss with 1/2 of the animals with a permanent mark. The data are simulated with tag-specific loss rates and dependence.

Format

A data frame with 1000 observations on the following 2 variables.

ch a character vector containing the encounter history of each bird

perm a factor variable indicating whether the animal is permanently marked ("yes") or not ("no")

Note

Data were simulated with the following code:

```
# simulate some double-tag data for 1000 animals

# over 5 cohorts and 6 sampling occasions; loss rate varies

# across tags and tag loss is dependent (beta=3)
set.seed(299811)
simdata=simHMM(data=data.frame(ch=c("++,0,0,0,0,0","0,++,0,0,0,0","0,0,++,0,0,0",
             "0,0,0,++,0,0","0,0,0,0,++,+-"),
                     freq=rep(1000/5,5)),model="hmmcjs2tl",
                       model.parameters=list(tau=list(formula=~tag1+tag2+tag1:tag2)),
                     initial=list(Phi=log(9),p=log(.666),tau=c(-2,-1,3)))

# treat every other animal as permanently marked;
# 00 observations are possible
simdata$perm=as.factor(rep(c("no","yes"),500))
# for non-permanently marked animals change
# "--" observations to "0" because
# they could not be detected
simdata$ch[simdata$perm=="no"]=gsub("--","0",simdata$ch[simdata$perm=="no"])
# save data
tagloss=simdata
save(tagloss,file="tagloss.rda")
```

Examples

```
# This example is excluded from testing to reduce package check time
# get data; the beta parameters used to simulate the data were
# Phi: 2.197, p: -0.4064  Tau:-2,-1,3
data(tagloss)
# process data with double-tag CJS model and use perm field to create groups;
# one half are permanently marked
dp=process.data(tagloss,model="hmmcjs2tl",groups="perm")
# create default design data
ddl=make.design.data(dp)
#set p=0 when the animal is not permanently marked if it goes to state 00 (both tags lost)
#which is equivalent to tag1=1 and tag2=1. The tag1, tag2 fields are created as part of default
#design data for p and tau.
#For p, there are 4 records for each occasion for each animal. The records are
#for the 4 possible tag loss states: 11,10,01,00. For the first occasion an animal was
#seen, p=1 (fix=1).
tail(ddl$p)
#For tau, there are 4 records for each occasion for each animal. The records are
#for the 4 possible tag loss states: 11,10,01,00. For state 11, tau=1 (fix=1) because
#a multinomial logit is used for this model. One of the values need to be fixed to 1
#to make the parameters identifiable because the probabilities sum to 1 for the
#transitions to the tag states.
tail(ddl$tau)
# For Phi there is a single record per occasion(interval) per animal.
tail(ddl$Phi)
# Animals that are not permanently marked cannot be seen in state 00 which is the
# same as tag1==1 & tag2==1
ddl$p$fix[ddl$p$perm=="no"&ddl$p$tag1==1&ddl$p$tag2==1]=0
# First fit a model assuming tag loss does not vary for the 2 tags. In that case,
# the probability is beta for a single loss and 2*beta for double loss. We get that
# model using I(tag1+tag2) which has values 0,1,2. Note that for the tau
# parameter the intercept is removed automatically. Also tag loss is independent in this
# model.
mod0=crm(dp,ddl,model.parameters=list(tau=list(formula=~I(tag1+tag2))),
        initial=list(Phi=2,p=.3,tau=c(-1)),hessian=TRUE)
# now fit a model allowing different loss rates for each tag but still independent
mod1=crm(dp,ddl,model.parameters=list(tau=list(formula=~tag1+tag2)),
        initial=list(Phi=2,p=.3,tau=c(-2,-1)),hessian=TRUE)
# now fit the model that was used to generate the data with dependence
mod2=crm(dp,ddl,model.parameters=list(tau=list(formula=~tag1+tag2+tag1:tag2)),
        initial=list(Phi=2,p=.3,tau=c(-2,-1,3)),hessian=TRUE)
# Now treat all as not permanently marked
tagloss$ch=gsub("--","0",tagloss$ch)
dp=process.data(tagloss,model="hmmcjs2tl")
ddl=make.design.data(dp)
ddl$p$fix[ddl$p$tag1==1&ddl$p$tag2==1]=0
mod3=crm(dp,ddl,model.parameters=list(tau=list(formula=~tag1+tag2)),
        initial=list(Phi=2,p=.3,tau=c(-2,-1)),hessian=TRUE)
# Model 2 is the best model but note that even though the tag loss model is
# incorrect in models 0 and 1 which assume independence, the survival estimate is
# only slightly less than for model 2. The model compensates by increasing the indiviudal
```

```
# tag loss rates to explain the observed 00's with the permanently marked animals.  Thus, if
# if you have enough marked animals you could end up with little bias in survival even if you
# have the wrong tag loss model.
mod0
mod1
mod2
# Note in model 3, even though tag-loss specific rates are estimated correctly
# survival is biased low because tag loss was dependent in simulating data
# but was assumed to be independent in fitted model and because there are no -- observations,
# the model assumes what are unobserved excess 00's are dead, so the survival estimate will be
# negatively biased. Note the data are different and AIC not comparable to other models.
mod3
if(require(expm))
{
   tag_status=function(k,x)
   {
    mat=t(sapply(1:k,function(k,x) (x%^%k)[1,] ,x=x))
    colnames(mat)=c("11","10","01","00","Dead")
    rownames(mat)=1:k
    return(mat)
    }
    par(mfrow=c(1,4))
    barplot(t(tag_status(4,mod0$results$mat$gamma[1,1,,])),
     beside=TRUE,ylim=c(0,1),main="mod0",legend.text=c("11","10","01","00","Dead"),
     args.legend=list(x=20,y=.9))
    barplot(t(tag_status(4,mod1$results$mat$gamma[1,1,,])),beside=TRUE,
                  ylim=c(0,1),main="mod1")
    barplot(t(tag_status(4,mod2$results$mat$gamma[1,1,,])),beside=TRUE,
                  ylim=c(0,1),main="mod2")
    barplot(t(tag_status(4,mod3$results$mat$gamma[1,1,,])),beside=TRUE,
                   ylim=c(0,1),main="mod3")
}
```

`valid.parameters` *Determine validity of parameters for a model (internal use)*

Description

Checks to make sure specified parameters are valid for a particular type of model.

Usage

```
valid.parameters(model, parameters)
```

Arguments

`model`	type of c-r model ("CJS", "Burnham" etc)
`parameters`	vector of parameter names (for example "Phi" or "p" or "S")

Value

Logical; TRUE if all parameters are acceptable and FALSE otherwise

Author(s)

Jeff Laake

See Also

setup.parameters, setup.model

Index