Package 'RMark'

October 6, 2014

Version 2.1.9

Date 2014-10-6

Title R Code for MARK Analysis

Author Jeff Laake <jeff.laake@noaa.gov> with code contributions from Eldar Rakhimberdiev, Ben Augustine and Daniel Turek, and example data and analysis from Bret Collier, Jay Rotella, David Pavlacky, and Andrew Paul.

Description This package provides an interface to the software package MARK developed by Gary White. MARK is freely available at (http://www.phidot.org/software/mark/download/) but is not open source.

SystemRequirements notepad.exe, mark.exe (>= 8.0) (or mark32.exe and mark64.exe) and rel_32.exe (see README.txt)

Depends R (>= 2.13.0)

Imports parallel, snowfall, matrixcalc, msm, coda

Suggests lattice, splines, nlme, plotrix

LazyLoad yes

License GPL (>= 2)

Maintainer Jeff Laake <Jeff.Laake@noaa.gov>

NeedsCompilation no

Repository CRAN

Date/Publication 2014-10-06 23:56:40

R topics documented:

ABeginnersGuide *A beginners introduction and guide to RMark*

Description

The RMark package is a collection of R functions that can be used as an interface to MARK for analysis of capture-recapture data.

Details

The library contains various functions that import/export capture data, build capture-recapture models, run the FORTRAN program MARK.EXE, and extract and display output. Program MARK has its own user interface; however, model development can be rather tedious and error-prone because the parameter structure and design matrix are created by hand. This interface in R was created to use the formula and design matrix functions in R to ease model development and reduce errors. This R interface has the following advantages: 1) Uses model notation to create design matrices rather than designing them by hand in MARK or in EXCEL, which makes model development faster and more reliable. All-different PIMS are automatically created for each group (if any). 2) Allows models based on group (factor variables) and individual covariates with groups created on the fly. Age, cohort, group and time variables are pre-defined for use in formulas. 3) Both real and beta labels are automatically added for easy output interpretation. 4) Input, output and specific results (eg parameter estimates, AICc etc) are stored in an R object where they can be manipulated as deemed useful (eg plotting, further calculations, simulation etc). 5) Parameter estimates can be displayed in triangular PIM format (if appropriate) for ease of interpretation. 6) Easy setup of batch jobs and the calls to the R functions document the model specifications and allow models to be easily reproduced or re-run if data are changed. 7) Covariate-specific estimates of real parameters can be computed within R without re-running the analysis.

The MARK capture-recapture models that are currently supported are provided in MarkModels.pdf which is installed in the RMark directory of your R library. You can also find a list in MARK under Help/Data Types. ' There is one limitation of this interface. All models in this interface are developed via a design matrix approach rather than coding the model structure via parameter index matrices (PIMS). In most cases, a logit or other link is used by default which has implications for ability of MARK to count the number of identifiable parameters (see `dipper` for an example). However, beginning with v1.7.6 the sin link is now supported if the formula specifies an identity design matrix for the parameter.

Before you begin, you must have installed MARK (`http://www.cnr.colostate.edu/~gwhite/mark/mark.htm`) on your computer or at least have a current copy of MARK.EXE. As long as you selected the default location for the MARK install (c:/Program Files/Mark), the `RMark` library will be able to find it. If for some reason, you choose to install it in a different location, see the note section in `mark` for instructions on setting the variable MarkPath to specify the path. In addition to installing MARK, you must have installed the `RMark` library into the R library directory. Once done with those tasks, run R and enter library(RMark) (or put it in your .First function) to attach the library of functions.

The following is a categorical listing of the functions in the library with a link to the help for each function. To start, read the help for functions `import.chdata` and `mark` to learn how to import your data and fit a simple model. The text files for the examples shown in `import.chdata` are

in the subdirectory data within the R Library directory in RMark. Next look at the example data sets and analyses dipper, edwards.eberhardt, and example.data. After you see the structure of the examples and the use of functions to fit a series of analyses, explore the remaining functions under Model Fitting, Batch Analyses, Model Selection and Summary and Display. If your data and models contain individual covariates, read the section on Real Parameter Computation to learn how to compute estimates of real parameters at various covariate values.

Input/Output data & results

import.chdata,read.mark.binary, extract.mark.output

Exporting Models to MARK interface

export.chdata, export.model

Model Fitting

mark, process.data, make.design.data, add.design.data, make.mark.model, run.mark.model merge_design.covariates

Batch analyses with functions

run.models, collect.models, create.model.list, mark.wrapper

Summary and display

summary.mark, print.mark, print.marklist, get.real, compute.real, print.summary.mark

Model Selection/Goodness of fit

adjust.chat, adjust.parameter.count, model.table , release.gof, model.average

Real Parameter computation

find.covariates, fill.covariates, compute.real , covariate.predictions

Utility and internal functions

collect.model.names, compute.design.data, extract.mark.output, inverse.link, deriv.inverse.link, setup.model, setup.parameters, valid.parameters, cleanup

For examples, see dipper for CJS and POPAN, see example.data for CJS with multiple grouping variables, see edwards.eberhardt for various closed-capture models, see mstrata for Multistrata, and see Blackduck for known fate. The latter two are examples of the use of mark.wrapper for a shortcut approach to creating a series of models. Other examples have been added for the various other models. In MarkModels.pdf it also lists the name of examples that are provided for each model.

Author(s)

Jeff Laake

References

MARK: Dr. Gary White, Department of Fishery and Wildlife Biology, Colorado State University, Fort Collins, Colorado, USA http://www.cnr.colostate.edu/~gwhite/mark/mark.htm

add.design.data *Add design data*

Description

Creates new design data fields in (ddl) that bin the fields cohort, age or time. Other fields (e.g., effort value for time) can be added to ddl with R commands.

Usage

```
add.design.data(data, ddl, parameter, type = "age", bins = NULL,
  name = NULL, replace = FALSE, right = TRUE)
```

Arguments

data	processed data list resulting from process.data
ddl	current design dataframe initially created with make.design.data
parameter	name of model parameter (e.g., "Phi" for CJS models)
type	either "age", "time" or "cohort"
bins	bins for grouping
name	name assigned to variable in design data
replace	if TRUE, replace any variable with same name as name
right	If TRUE, bin intervals are closed on the right

Details

Design data can be added to the parameter specific design dataframes with R commands. Often the additional fields will be functions of cohort, age or time. add.design.data provides an easy way to add fields that bin (put into intervals) the original values of cohort, age or time. For example, age may have levels from 0 to 10 which means the formula ~age will have 11 parameters, one for each level of the factor. It might be more desirable and more parimonious to have a simpler 2 age class model of young and adults. This can be done easily by adding a new design data field that bins age into 2 intervals (age 0 and 1+) as in the following example:

```
 ddl=make.design.data(proc.example.data)
ddl=add.design.data(proc.example.data,ddl,parameter="Phi",type="age",
bins=c(0,.5,10),name="2ages")
```

By default, the bins are open on the left and closed on the right (i.e., binning x by (x1,x2] is equivalent to x1<x<=x2) except for the first interval which is closed on the left. Thus, for the above example, the age bins are [0,.5] and (.5,10]. Since the ages in the example are 0,1,2... using any value >0 and <1 in place of 0.5 would bin the ages into 2 classes of 0 and 1+. This behavior can be modified by changing the argument right=FALSE to create an interval that is closed on the left and open on the right. In some cases this can make reading the values of the levels somewhat easier. It is important to recognize that the new variable is only added to the design data for the

defined parameter and can only be used in model formula for that parameter. Multiple calls to add.design.data can be used to add the same or different fields for the various parameters in the model. For example, the same 2 age class variable can be added to the design data for p with the command:

```
ddl=add.design.data(proc.example.data,ddl,parameter="p",type="age",
bins=c(0,.5,10),name="2ages")
```

The name must be unique within the parameter design data, so they should not use pre-defined values of group, age, Age, time, Time,cohort, Cohort. If you choose a name that already exists in the design data for the parameter, it will not be added but it can replace the variable if replace=TRUE. For example, the 2ages variable can be re-defined to use 0-1 and 2+ with the command:

```
ddl=add.design.data(proc.example.data,ddl,parameter="Phi",type="age",
bins=c(0,1,10),name="2ages",replace=TRUE)
```

Keep in mind that design data are stored with the mark model object so if a variable is redefined, as above, this could become confusing if some models have already been constructed using a different definition for the variable. The model formula and names would appear to be identical but they would have a different model structure. The difference would be apparent if you examined the design data and design matrix of the model object but would the difference would be transparent based on the model names and formula. Thus, it would be best to avoid constructing models from design data fields with different structures but the same name.

Value

Design data list with new field added for the specified parameter. See make.design.data for a description of the list structure.

Note

For the specific case of "closed" capture models, the parameters p (capture probability) and c (recapture probability) can be treated in a special fashion. Because they really the same type of parameter, it is useful to be able to share a common model structure (i.e., same columns in the design matrix). This is indicated with the share=TRUE element in the model description for p. If the parameters are shared then the additional covariate c is added to the design data, which is c=0 for parameter p and c=1 for parameter c. This enables an additive model to be developed where recapture probabilities mimic the pattern in capture probabilities except for an additive constant. The covariate c can only be used in the model for p if share=TRUE. If the latter is not set using c in a formula will result in an error. Likewise, if share=TRUE, then the design data for p and c must be the same because the design data are merged in constructing the design matrix. Thus if you add design data for parameter p, you should add a similar field for parameter c if you intend to fit shared models for the two parameters. If the design data do not match and you try to fit a shared model, an error will result.

Author(s)

Jeff Laake

See Also

make.design.data, process.data

Examples

```
data(example.data)
example.data.proc=process.data(example.data)
ddl=make.design.data(example.data.proc)
ddl=add.design.data(example.data.proc,ddl,parameter="Phi",type="age",
  bins=c(0,.5,10),name="2ages")
ddl=add.design.data(example.data.proc,ddl,parameter="p",type="age",
bins=c(0,.5,10),name="2ages")
ddl=add.design.data(example.data.proc,ddl,parameter="Phi",type="age",
bins=c(0,1,10),name="2ages",replace=TRUE)
```

---

adjust.parameter.count

Adjust count of estimated parameters

Description

Modifies number of estimated parameters and the resulting AICc value for model selection.

Usage

```
adjust.parameter.count(model, npar)
```

Arguments

model	MARK model object
npar	Value of count of estimated parameters

Details

When a model is run the parameter count determined by MARK is stored in results$npar and the AICc value is stored in results$AICc. If the argument adjust is set to TRUE in the call to run.mark.model and MARK determined that the design matrix was not full rank (i.e., the parameter count is less than the columns of the design matrix), then the parameter count from MARK is stored in results$npar.unadjusted and AICc in results$AICc.unadjusted and results$npar is set to the number of columns of the design matrix and results$AICc uses the assumed full rank value of npar. This function allows the parameter count to be reset to any value less than or equal to the number of columns in the design matrix. If results$npar.unadjusted exists it is kept as is. If it doesn't exist, then the current values of results$npar and results$AICc are stored in the .unadjusted fields to maintain the values from MARK, and the new adjusted values defined by the function argument npar are stored in results$npar and results$AICc. In the example below, the CJS model Phi(t)p(t) is fitted with the call to mark which defaults to adjust=TRUE.

This is used to show how `adjust.parameter.count` can be used to adjust the count to 11 from the full rank count of 12. Alternatively, the argument `adjust=FALSE` can be added to prevent the adjustment which is appropriate in this case because Phi(6) and p(6) are confounded.

Value

model: the mark model object with the adjustments made

Author(s)

Jeff Laake

See Also

`run.mark.model`,`model.table`

Examples

```
data(dipper)
ptime=list(formula=~time)
Phitime=list(formula=~time)
dipper.phitime.ptime=mark(dipper,model.parameters=list(Phi=Phitime, p=ptime))
dipper.phitime.ptime=adjust.parameter.count(dipper.phitime.ptime,11)
dipper.phitime.ptime=mark(dipper,model.parameters=list(Phi=Phitime, p=ptime),
                          adjust=FALSE)
```

---

adjust.value *Adjust over-dispersion scale or a result value such as effective sample size*

Description

Adjust value of over-dispersion constant or another result value for a collection of models which modifies model selection criterion and estimated standard errors.

Usage

```
adjust.value(field="n",value,model.list)
       adjust.chat(chat=1,model.list)
```

Arguments

`field`	Character string containing name of the field; either `chat` or a field in `model$results` such as `n` for sample size used in AICc or QAICc
`value`	new value for field

model.list	marklist created by the function collect.models which has each model object and a model.table at the end. For the entire collection of models each chat is adjusted. If the argument type is specified the collected models are limited to mark analyses with that specific type of model ("CJS")
chat	Over-dispersion scale

Details

The value of chat is stored with the model object except when there is no over-dispersion (chat=1). This function assigns a new value of chat for the collection of models specified by model.list and/or type. The value of chat is used by model.table for model selection in computing QAICc unless chat=1. It is also used in summary.mark, get.real and compute.real to adjust standard errors and confidence intervals. Note that the standard errors and confidence intervals in results$beta,results$beta.vcv results$real, results$derived and results$derived.vcv are not modified and always assume chat=1.

It can also be used to modify a field in model$results such as n which is ESS (effective sample size) from MARK output that is used in AICc/QAICc calculations.

Value

model.list with all models given the new chat value and model.table adjusted for chat values

Note

See note in collect.models

Author(s)

Jeff Laake

See Also

model.table, summary.mark, get.real ,compute.real

Examples

```
#
# The following are examples only to demonstrate selecting different
# model sets for adjusting chat and showing model selection table.
# It is not a realistic analysis.
#

data(dipper)
mod1=mark(dipper)
mod2=mark(dipper,model.parameters=list(Phi=list(formula=~time)))
mod3=mark(dipper,model="POPAN",initial=1)
cjs.results=collect.models(type="CJS")
cjs.results  # show model selection results for "CJS" models
POPAN.results=collect.models(type="POPAN")
POPAN.results # show model selection results for "POPAN" model
```

```
# adjust chat for all models to 2
cjs.results=adjust.chat(2,cjs.results)
cjs.results
# adjust chat for all models to 2
POPAN.results=adjust.chat(2,POPAN.results)
POPAN.results
```

`Blackduck` *Black duck known fate data*

Description

A known fate data set on Black ducks that accompanies MARK as an example analysis using the Known model.

Format

A data frame with 48 observations on the following 5 variables.

ch a character vector containing the encounter history of each bird

BirdAge the age of the bird: a factor with levels `0` `1` for young and adult

Weight the weight of the bird at time of marking

Wing_Len the wing-length of the bird at time of marking

condix the condition index of the bird at time of marking

Details

This is a data set that accompanies program MARK as an example for Known fate. The data can be stratified using BirdAge as a grouping variable. The function `run.Blackduck` defined below in the examples creates some of the models used in the dbf file that accompanies MARK.

Note that in the MARK example the variable is named Age. In the R code, the fields "age" and "Age" have specific meanings in the design data related to time since release. These will override the use of a field with the same name in the individual covariate data, so the names "time", "Time", "cohort", "Cohort", "age", and "Age" should not be used in the individual covariate data with possibly the exception of "cohort" which is not defined for models with "Square" PIMS such as POPAN and other Jolly-Seber type models.

Examples

```
data(Blackduck)
# Change BirdAge to numeric; starting with version 1.6.3 factor variables are
# no longer allowed.  They can work as in this example but they can be misleading
# and fail if the levels are non-numeric.  The real parameters will remain
# unchanged but the betas will be different.
Blackduck$BirdAge=as.numeric(Blackduck$BirdAge)-1
run.Blackduck=function()
```

```
{
#
# Process data
#
bduck.processed=process.data(Blackduck,model="Known")
#
# Create default design data
#
bduck.ddl=make.design.data(bduck.processed)
#
#  Add occasion specific data min < 0; I have no idea what it is
#
bduck.ddl$S$min=c(4,6,7,7,7,6,5,5)
#
#  Define range of models for S
#
S.dot=list(formula=~1)
S.time=list(formula=~time)
S.min=list(formula=~min)
S.BirdAge=list(formula=~BirdAge)
#
# Note that in the following model in the MARK example, the covariates
# have been standardized.  That means that the beta parameters will be different
# for BirdAge, Weight and their interaction but the likelihood and real parameter
# estimates are the same.
#
S.BirdAgexWeight.min=list(formula=~min+BirdAge*Weight)
S.BirdAge.Weight=list(formula=~BirdAge+Weight)
#
# Create model list and run assortment of models
#
model.list=create.model.list("Known")
bduck.results=mark.wrapper(model.list,data=bduck.processed,ddl=bduck.ddl,
               invisible=FALSE,threads=2)

#
# Return model table and list of models
#
return(bduck.results)
}
bduck.results=run.Blackduck()
bduck.results
```

---

brownie *San Luis Valley mallard data*

Description

A recovery data set for mallards in San Luis Valley Colorado

Format

A data frame with 108 observations on the following 5 variables.

ch a character vector containing the encounter history of each bird

freq frequency of that encounter history

ReleaseAge the age of the bird when it was released

Details

This is a data set that accompanies program MARK as an example for Brownie and Seber recovery model. In those input files it is in a summarized format. Here it is in the LD encounter history format. The data can be stratified using ReleaseAge (Adult, Young) as a grouping variable.

Note that in the MARK example the variable is named Age. In the R code, the fields "age" and "Age" have specific meanings in the design data related to time since release. These will override the use of a field with the same name in the individual covariate data, so the names "time", "Time", "cohort", "Cohort", "age", and "Age" should not be used in the individual covariate data with possibly the exception of "cohort" which is not defined for models with "Square" PIMS such as POPAN and other Jolly-Seber type models.

Examples

```
# brownie=import.chdata("brownie.inp",field.types=c("n","f"))

 data(brownie)
# default ordering of ReleaseAge is alphabetic so it is
# Adult, Young which is why initial.ages=c(1,0)
# Seber Recovery
br=process.data(brownie,model="Recovery",groups="ReleaseAge",age.var=1,initial.ages=c(1,0))
br.ddl=make.design.data(br,parameters=list(S=list(age.bins=c(0,1,10)),
                                           r=list(age.bins=c(0,1,10))),right=FALSE)
mod=mark(br,br.ddl,model.parameters=list(S=list(formula=~-1+age:time,link="sin"),
                                           r=list(formula=~-1+age:time,link="sin")))
# Brownie Recovery
br=process.data(brownie,model="Brownie",groups="ReleaseAge",age.var=1,initial.ages=c(1,0))
br.ddl=make.design.data(br,parameters=list(S=list(age.bins=c(0,1,10)),
                                f=list(age.bins=c(0,1,10))),right=FALSE)
mod=mark(br,br.ddl,model.parameters=list(S=list(formula=~-1+age:time,link="sin"),
                                f=list(formula=~-1+age:time,link="sin")))
mod=mark(br,br.ddl,model.parameters=list(S=list(formula=~-1+age,link="sin"),
                                f=list(formula=~-1+age,link="sin")))
#Random effects Seber recovery
br=process.data(brownie,model="REDead",groups="ReleaseAge",age.var=1,initial.ages=c(1,0))
br.ddl=make.design.data(br,parameters=list(S=list(age.bins=c(0,1,10)),
                                      r=list(age.bins=c(0,1,10))),right=FALSE)
mod=mark(br,br.ddl,model.parameters=list(S=list(formula=~age),r=list(formula=~age)))
#Pledger Mixture Seber recovery
br=process.data(brownie,model="PMDead",groups="ReleaseAge",
                           mixtures=3,age.var=1,initial.ages=c(1,0))
br.ddl=make.design.data(br,parameters=list(S=list(age.bins=c(0,1,10)),
                            r=list(age.bins=c(0,1,10))),right=FALSE)
```

```
mod=mark(br,br.ddl,model.parameters=list(pi=list(formula=~mixture),
                     S=list(formula=~age+mixture),r=list(formula=~age)))
br=process.data(brownie,model="PMDead",groups="ReleaseAge",
                     mixtures=2,age.var=1,initial.ages=c(1,0))
br.ddl=make.design.data(br,parameters=list(S=list(age.bins=c(0,1,10)),
                      r=list(age.bins=c(0,1,10))),right=FALSE)
mod=mark(br,br.ddl,model.parameters=list(pi=list(formula=~age),
                      S=list(formula=~age+mixture),r=list(formula=~age)))
```

`cleanup` *Removes unused MARK output files*

Description

Identifies all unused (orphaned) mark*.inp, .vcv, .res and .out and .tmp files in the working directory and removes them. The orphaned files are determined by examining all mark objects and lists of mark objects (created by `collect.models`) to create a list of files in use. All other files are treated as orphans to delete.

Usage

```
cleanup(lx = NULL, ask = TRUE, prefix = "mark")
```

Arguments

`lx`	listing of R objects; defaults to list of workspace from calling environment; if NULL it uses ls(envir=parent.frame())
`ask`	if TRUE, prompt whether each file should be removed. Typically will be used with ask=FALSE but default of TRUE may avoid problems
`prefix`	prefix for filename if different than "mark"

Details

This function removes orphaned output files from MARK. This occurs when there are output files in the subdirectory that are not associated with a mark object in the current R session (.Rdata). For example, if you repeat an analysis or set of analyses and store them in the same object then the original set of output files would no longer be linked to an R object and would be orphaned.

As an example, consider the `mallard` analysis. The first time you run the analysis script in an empty subdirectory it would create 9 sets of MARK output files (mark001.out,.vcv,.res,.inp to mark009.out,.vcv,.res,.inp) and each would be linked to one of the objects in `nest.results`. When the command `AgePpnGrass=nest.results$AgePpnGrass` was issued, both of those `mark` objects were linked to the same set of output files. Now if you were to repeat the above commands and re-run the models and stored the results into `nest.results` again, it would create files with prefixes 10 to 18. Because that would have replaced `nest.results`, none of the files numbered 1 to 9 would be linked to an object. `cleanup(ask=FALSE)` automatically removes those orphan files. If you delete all objects in the R session with the command `rm(list=ls(all=TRUE))`, then subsequently

cleanup(ask=FALSE) will delete all MARK output files because all of them will be orphans. Output files can also become orphans if MARK finishes but for some reason R crashes or you forget to save your session before you exit R. Orphan output files can be re-linked to an R object without re-running MARK by re-running the mark function in R and specifing the filename argument to match the base portion of the orphaned output file (eg "mark067"). It will create all of the necessary R objects and then asks if you want to use the existing file. If you respond affirmatively it will link to the orphaned files.

Value

None

Author(s)

Jeff Laake

`collect.model.names` *Collect names of MARK model objects from list of R objects (internal function)*

Description

Either names of all mark model objects (type=NULL) or names of mark model objects of a specific type (type) are extracted from a vector of R objects (lx) that was collected from the parent environment (frame) of the function that calls collect.model.names. Thus, it is two frames back (parent.frame(2)).

Usage

```
collect.model.names(lx, type = NULL, warning = TRUE)
```

Arguments

lx	vector of R object names from parent.frame(2)
type	either NULL (for all types) or a character model type (eg "CJS")
warning	if TRUE warning given when models of different types are collected

Details

If type=NULL then the names of all objects of class(x)[1]="mark" in lx are returned. If type is specified, then the names of all objects of class(x)=c("mark",type) in lx are returned.

This function was written with the intention that it would be called from other functions (e.g., collect.models, run.models) but it will work if called directly (e.g., collect.model.names(lx=ls())). While this function returns a vector of model names, collect.models returns a list of model objects. The latter can be used to easily create a list of models created in a function to be used as a return value without listing all the names of the functions. It uses collect.model.names to perform that function.

Value

model.list: a vector of mark model names

Author(s)

Jeff Laake

See Also

collect.models, run.models, model.table

---

collect.models *Collect MARK models into a list and optionally construct a table of model results*

Description

Collects mark models contained in lx of specified type (if any) and returns models in a list with a table of model results if table=TRUE.

Usage

```
collect.models(lx = NULL, type = NULL, table = TRUE, adjust = TRUE,
  external = FALSE)
```

Arguments

lx	if NULL, constructs vector of object names (ls()) from frame of calling function otherwise it uses specifed names in lx
type	either NULL (for all types) or a character model type (e.g. type="CJS")
table	if TRUE, a table of model results is also included in the returned list
adjust	if TRUE, adjusts number of parameters (npar) to number of columns in design matrix which modifies AIC
external	if TRUE the mark objects are saved externally rather than in the resulting marklist; the filename for each object is kept in its place

Details

If lx is NULL a vector of object names in the parent frame is constructed for lx. Within lx all mark model objects (i.e., class(x)[1]=="mark") are returned if type is NULL. If type is specified and is valid, then the names of all mark model objects of the specified type (i.e., class(x) = c("mark",type)) in lx are returned. If table=TRUE a table of model selection results is also included in the returned list.

This function was written to be able to easily collect a series of mark models in a list without specifying the names of each model object. This is useful in constructing a return value of a function that runs a series of models for a particular analysis. For an example see dipper.

Value

model.list: a list of mark models and optionally a table of model results.

Note

This function and others that use it or use collect.model.names to collect a series of models or assign a value to a series of models (e.g., adjust.chat) should be used with a degree of caution. It is important to understand the scope of the collection. If the call to this function is made at the R prompt, then it will collect all models (of a particular type if any) within the current .Rdata file. If the call to this function (or one like it) is called from within a function that runs a series of analyses then the collection is limited to the function frame (i.e., only models defined within the function). Thus, it is wise to either use a different .Rdata file for each data set (e.g., one for dipper, another for edwards.eberhardt, etc) or to run everything within functions as illustrated by dipper or edwards.eberhardt. Using a separate .Rdata file is equivalent to having separate .DBF/.FPT files with MARK. It is important to note that functions such as adjust.chat will adjust the value of chat across analyses unless specifically given a list of models to adjust.

Author(s)

Jeff Laake

See Also

merge.mark,remove.mark, collect.model.names,run.models,model.table,dipper

Examples

```
# see examples in dipper, edwards.eberhardt and example.data
```

---

compute.design.data *Compute design data for a specific parameter in the MARK model (internal use)*

Description

For a specific type of parameter (e.g., Phi, p, r etc), it creates a data frame containing design data for each parameter of that type in the model as structured by an all different PIM (parameter information matrix). The design data are used in constructing the design matrix for MARK with user-specified model formulae as in make.mark.model.

Usage

```
compute.design.data(data, begin, num, type = "Triang", mix = FALSE,
  rows = 0, pim.type = "all", secondary, nstrata = 1, tostrata = FALSE,
  strata.labels = NULL, subtract.stratum = strata.labels,
  common.zero = FALSE, sub.stratum = 0, limits = NULL)
```

Arguments

data	data list created by process.data
begin	0 for survival type, 1 for capture type
num	number of parameters relative to number of occasions (0 or -1)
type	type of parameter structure (Triang (STriang) or Square)
mix	if TRUE this is a mixed parameter
rows	number of rows relative to number of mixtures
pim.type	type of pim structure; either all (all-different) or time
secondary	TRUE if a parameter for the secondary periods of robust design
nstrata	number of strata for multistrata
tostrata	set to TRUE for transition parameters
strata.labels	labels for strata as identified in capture history
subtract.stratum	for each stratum, the to.strata that is computed by subtraction
common.zero	if TRUE, uses a common begin.time to set origin (0) for Time variable defaults to FALSE for legacy reasons but should be set to TRUE for models that share formula like p and c with the Time model
sub.stratum	the number of strata to subtract for parameters that use mlogit across strata like pi and Omega for RDMSOpenMisClass
limits	For RDMSOccRepro values that set row and col (if any) start on states

Details

This function is called by make.design.data to create all of the default design data for a particular type of model and by add.design.data to add binned design data fields for a particular type of parameter. The design data created by this function include group, age, time and cohort as factors variables and continuous (non-factor) versions of all but group. In addition, if groups have been defined for the data, then a data column is added for each factor variable used to define the groups. Also for specific closed capture heterogeneity models (model="HetClosed", "FullHet", "HetHug", "FullHetHug") the data column mixture is added to the design data. The arguments for this function are defined for each model by the function setup.model.

Value

design.data: a data frame containing all of the design data fields for a particular type of parameter

group	group factor level
age	age factor level
time	time factor level
cohort	cohort factor level
Age	age as a continuous variable
Time	time as a continuous variable
Cohort	cohort as a continuous variable
mixture	mixture factor level
other fields	any factor variables used to define groups

Author(s)

Jeff Laake

See Also

make.design.data, add.design.data

`compute.link` *Compute estimates of link values*

Description

Computes link values (design*beta) for real parameters, and var-cov from design matrix (design) and coefficients (beta)

Usage

```
compute.link(model, beta = NULL, design = NULL, data = NULL,
  parm.indices = NULL, vcv = TRUE)
```

Arguments

model	MARK model object
beta	Estimates of beta parameters
design	numeric design matrix for MARK model with any covariate values filled in
data	dataframe with covariate values that are averaged for estimates
parm.indices	index numbers from PIMS for rows in design matrix to use
vcv	logical; if TRUE, returns v-c matrix of link values

Details

This function is very similar to compute.real except that it provides estimates of link values before they are transformed to real estimates using the inverse-link. It is called by get.link to make calculations but can be called separately. The value is always a dataframe for the estimates and design data and optionally a variance-covariance matrix. See get.real for further details about the arguments.

Value

estimates: If vcv=TRUE, a list is returned with elements vcv.link and the dataframe estimates. If vcv=FALSE, only the estimates dataframe is returned which has the same structure as in get.real.

Author(s)

Jeff Laake

See Also

get.link

---

compute.links.from.reals

Compute link values from real parameters

Description

Computes link values from reals using 1-1 real to beta(=link) transformation. Also, creates a v-c matrix for the link values if vcv.real is specified.

Usage

```
compute.links.from.reals(x, model, parm.indices = NULL, vcv.real = NULL,
  use.mlogits = TRUE)
```

Arguments

x	vector of real estimates to be converted to link values
model	MARK model object used only to obtain model structure/links etc. If function is being called for model averaged estimates, then any model in the model list used to construct the estimates is sufficient
parm.indices	index numbers from PIMS for rows in design matrix(non-simplified indices); x[parm.indices] are computed
vcv.real	v-c matrix for the real parameters
use.mlogits	logical; if FALSE then parameters with mlogit links are transformed with logit rather than mlogit for creating confidence intervals for each value

Details

It has 2 uses both related to model averaged estimates. Firstly, it is used to transform model averaged estimates so the normal confidence interval can be constructed on the link values and then back-transformed to real space. The second function is to enable parametric bootstrapping in which the error distbution is assumed to be multivariate normal for the link values. From a single model, the link values are easily constructed from the betas and design matrix so this function is not needed. But for model averaging there is no equivalent because the real parameters are averaged over a variety of models with the same real parameter structure but differing design structures. This function allows for link values and their var-cov matrix to be created from the model averaged real estimates.

Value

A list with the estimates (link values) and the links that were used. If vcv.real = TRUE, then the v-c matrix of the links is also returned.

Author(s)

Jeff Laake

See Also

model.average

---

compute.real *Compute estimates of real parameters*

Description

Computes real estimates and var-cov from design matrix (design) and coefficients (beta) using specified link functions

Usage

```
compute.real(model, beta = NULL, design = NULL, data = NULL, se = TRUE,
  vcv = FALSE)
```

Arguments

model	MARK model object
beta	estimates of beta parameters for real parameter computation
design	design matrix for MARK model
data	dataframe with covariate values that are averaged for estimates
se	if TRUE returns std errors and confidence interval of real estimates
vcv	logical; if TRUE, sets se=TRUE and returns v-c matrix of real estimates

Details

The estimated real parameters can be derived from the estimated beta parameters, a completed design matrix, and the link function specifications. MARK produces estimates of the real parameters, se and confidence intervals but there are at least 2 situations in which it is useful to be able to compute them after running the analysis in MARK: 1) adjusting confidence intervals for estimated over-dispersion, and 2) making estimates for specific values of covariates. The first case is done in get.real with a call to this function. It is done by adjusting the estimated standard error of the beta parameters by multiplying it by the square root of chat to adjust for over-dispersion. A normal 95 +/- 1.96*se) and this is then back-transformed to the real parameters using inverse.link with the appropriate inverse link function for the parameter to construct a 95 There is one exception. For parameters using the mlogit transformation, a logit transformation of each individual real Psi and its se are used to derive the confidence interval. The estimated standard error for the real parameter is also scaled by the square root of the over-dispersion constant chat stored in model$chat. But, the code actually computes the variance-covariance matrix rather than relying on the values from the MARK output because real estimates will depend on any individual covariate values used in the model which is the second reason for this function.

New values of the real parameter estimates can easily be computed by simply changing the values of the covariate values in the design matrix and computing the inverse-link function using the beta parameter estimates. The covariate values to be used can be specified in one of 2 ways. 1) Prior to making a call to this function, use the functions `find.covariates` to extract the rows of the design matrix with covariate values and either fill in those values aautomatically with the options provided by `find.covariates` or edit those values to be the ones you want and then use `fill.covariates` to replace the values into the design matrix and use it as the value for the argument `design`, or 2) automate this step by specifying a value for the argument `data` which is used to take averages of the covariate values to fill in the covariate entries of the design matrix. In computing real parameter estimates from individual covariate values it is important to consider the scale of the individual covariates. By default, an analysis with MARK will standardize covariates by subtracting the mean and dividing by the standard deviation of the covariate value. However, in the `RMark` library all calls to MARK.EXE do not standardize the covariates and request real parameter estimates based on the mean covariate values. This was done because there are many instances in which it is not wise to use the standardization implemented in MARK and it is easy to perform any standardization of the covariates with R commands prior to fitting the models. Also, with pre-standardized covariates there is no confusion in specifying covariate values for computation of real estimates. If the model contains covariates and the argument `design` is not specified, the design matrix is extracted from `model` and all individual covariate values are assigned their mean value to be consistent with the default in the MARK analysis.

If a value for `beta` is given, those values are used in place of the estimates `model$results$beta$estimate`.

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 se=TRUE or vcv=TRUE it also contains standard errors and confidence intervals and notation of whether parameters are fixed or at a boundary
`vcv.real`	variance-covariance matrix of real estimates

Author(s)

Jeff Laake

See Also

`get.real`,`fill.covariates`,`find.covariates`,`inverse.link`,`deriv_inverse.link`

Examples

```
# see examples in fill.covariates
```

convert.inp *Convert MARK input file to RMark dataframe*

Description

Converts encounter history inp files used to create a MARK project into a dataframe for use with RMark. Group structure in frequencies is converted to factor variables that can be used to create groups in RMark. Covariates are copied straight across. Only works with encounter history format for input files and not specialized ones for known-fate or Brownie models.

Usage

```
convert.inp(inp.filename, group.df = NULL, covariates = NULL,
  use.comments = FALSE)
```

Arguments

inp.filename	name of input file; inp extension is assumed and does not need to be specified
group.df	dataframe with grouping variables that contains a row for each group defined in the input file row1=group1, row2=group2 etc. Names and number of columns in the dataframe is set by user to define grouping variables in RMark dataframe
covariates	names to be assigned to the covariates defined in the inp file
use.comments	if TRUE values within /* and */ on data lines are used as row.names for the RMark dataframe. Only use this option if they are unique values.

Details

The encounter history format for MARK is structured as follows: capture (encounter) history, followed by a frequency field for each group, followed by any covariates and then a semi-colon at the end of the line. Comments are allowed within /* and */. The RMark format is a dataframe with a different structure. Each record(row) in the dataframe is for one or more animals within a single group and if there is group structure then the dataframe contains factor variables that can be used to create groups. For example, the following is a little snippet of the same data with 2 groups Males/Females and a covariate weight in the two different formats:

```
 MARK encounter history file (in make believe test.inp): 1001
1 0 10; 1101 0 2 5; 0101 3 1 6;

RMark dataframe: ch freq sex weight 1001 1 M 10 1101 2 F 5 0101 3 M 6 0101 1
F 6
```

To convert from the MARK format to the RMark format it is necessary to define the variables used to define the groups (if any) and to define the covariate field names (if any). For the example above, if test.inp is in the same directory as the current working directory, the call would be:

```
test = convert.inp("test",group.df=data.frame(sex=c("M","F")),
covariates="weight")
```

Comments spanning lines in the .inp file are ignored and deleted as are blank lines. If each line has a unique identifier in the comments then by setting `use.comments=TRUE`, the text of the comment (e.g.,tag number) will be assigned as the row name in the RMark dataframe. This will only work if each line only represents a single animal or a set of animals in a single group. If file was structured as follows:

```
 MARK encounter history file (in make believe test.inp): 1001
1 0 10 /*1*/; 1101 0 2 5 /*2*/; 0101 3 1 6 /*3*/;
```

an error would occur

```
 Error in convert.inp("test", group.df = data.frame(sex =
c("M", "F")),: Row names not unique. Set use.comments to default value FALSE
```

because it would try to use "3" as the row name for the 3 males and the 1 female represented by the last row.

The extension .inp is optional for files with that extension. If the file has a different extension the entire filename must be specified.

Note that there are limitations to this function. You cannot have extra blank lines in the file, the number of columns (tab, space or comma delimited) must be the same in each line unless the line is just a comment line /* */. In the latter case, the /* must begin the line and the */ must end the line with no extra characters (blanks included) in before or after.

Value

Dataframe with fields ch(character encounter history), freq (frequency of encounter history), followed by grouping variables (if any) and then covariates (if any)

Author(s)

Jeff Laake

See Also

process.data

Examples

```
# MARK example input file
pathtodata=paste(path.package("RMark"),"extdata",sep="/")
dipper=convert.inp(paste(pathtodata,"dipper",sep="/"),
                   group.df=data.frame(sex=c("M","F")))
# Example input files that accompany the MARK electronic book
#  (http://www.phidot.org/software/mark/docs/book/)
bd=convert.inp(paste(pathtodata,"blckduck",sep="/"),
         covariates=c("age","weight","winglen","ci"),use.comments=TRUE)
aa=convert.inp(paste(pathtodata,"aa",sep="/"),
      group.df=data.frame(sex=c("Poor","Good")))
adult=convert.inp(paste(pathtodata,"adult",sep="/"))
```

```
age=convert.inp(paste(pathtodata,"age",sep="/"))
age_ya=convert.inp(paste(pathtodata,"age_ya",sep="/"),
      group.df=data.frame(age=c("Young","Adult")))
capsid=convert.inp(paste(pathtodata,"capsid",sep="/"))
clogit_demo=convert.inp(paste(pathtodata,"clogit_demo",sep="/"))
deer=convert.inp(paste(pathtodata,"deer",sep="/"))
ed_males=convert.inp(paste(pathtodata,"ed_males",sep="/"))
F_age=convert.inp(paste(pathtodata,"F_age",sep="/"))
indcov1=convert.inp(paste(pathtodata,"indcov1",sep="/"),
         covariates=c("cov1","cov2"))
indcov2=convert.inp(paste(pathtodata,"indcov2",sep="/"),
          covariates=c("cov1","cov2"))
island=convert.inp(paste(pathtodata,"island",sep="/"))
linear=convert.inp(paste(pathtodata,"linear",sep="/"))
young=convert.inp(paste(pathtodata,"young",sep="/"))
transient=convert.inp(paste(pathtodata,"transient",sep="/"))
ms_gof=convert.inp(paste(pathtodata,"ms_gof",sep="/"))
m_age=convert.inp(paste(pathtodata,"m_age",sep="/"))
ms_cjs=convert.inp(paste(pathtodata,"ms_cjs",sep="/"))
ms_directional=convert.inp(paste(pathtodata,"ms_directional",sep="/"))
ed=convert.inp(paste(pathtodata,"ed",sep="/"),
            group.df=data.frame(sex=c("Male","Female")))
multigroup=convert.inp(paste(pathtodata,"multi_group",sep="/"),
            group.df=data.frame(sex=c(rep("Female",2),rep("Male",2)),
            Colony=rep(c("Good","Poor"),2)))
LD1=convert.inp(paste(pathtodata,"LD1",sep="/"),
           group.df=data.frame(age=c("Young","Adult")))
yngadt=convert.inp(paste(pathtodata,"yngadt",sep="/"),
            group.df=data.frame(age=c("Young","Adult")))
effect_size=convert.inp(paste(pathtodata,"effect_size",sep="/"),
             group.df=data.frame(colony=c("Poor","Good")))
effect_size3=convert.inp(paste(pathtodata,"effect_size3",sep="/"),
             group.df=data.frame(colony=c("1","2","3")))
```

`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`	MARK 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

---

covariate.predictions *Compute estimates of real parameters for multiple covariate values*

Description

Computes real estimates for a dataframe of covariate values and the var-cov matrix of the real estimates.

Usage

```
covariate.predictions(model, data = NULL, indices = NULL, drop = TRUE,
  revised = TRUE, mata = FALSE, normal.lm = FALSE, residual.dfs = 0,
  alpha = 0.025, ...)
```

Arguments

model	MARK model object or marklist
data	dataframe with covariate values used for estimates; if it contains a field called index the covariates in each row are only applied to the parameter with that index and the argument indices is not needed
indices	a vector of indices from the all-different PIM structure for parameters to be computed

drop	if TRUE, models with any non-positive variance for betas are dropped
revised	if TRUE it uses eq 6.12 from Burnham and Anderson (2002) for model averaged se; otherwise it uses eq 4.9
mata	if TRUE, create model averaged tail area confidence intervals as described by Turek and Fletcher
normal.lm	Specify normal.lm=TRUE for the normal linear model case, and normal.lm=FALSE otherwise. When normal.lm=TRUE, the argument 'residual.dfs' must also be supplied. See USAGE section, and Turek and Fletcher (2012) for additional details.
residual.dfs	A vector containing the residual (error) degrees of freedom under each candidate model. This argument must be provided when the argument normal.lm=TRUE.
alpha	The desired lower and upper error rate. Specifying alpha=0.025 corresponds to a 95 alpha=0.05 to a 90 Default value is alpha=0.025.
...	additional arguments passed to specific functions

Details

This function has a similar use as compute.real except that it is specifically designed to compute real parameter estimates for multiple covariate values for either a single model or to compute model averaged estimates across a range of models within a marklist. This is particularly useful for computing and plotting the real parameter as a function of the covariate with pointwise confidence bands (see example below). The function also computes a variance-covariance matrix for the real parameters. For example, assume you had a model with two age classes of young and adult and survial for young was a function of weight and you wanted to estimate survivorship to some adult age as a function of weight. To do that you need the survival for young as a function of weight, the adult survival, the variance of each and their covariance. This function will allow you to accomplish tasks like these and many others.

When a variance-covariance matrix is computed for the real parameters, it can get too large for available memory for a large set of real parameters. Most models contain many possible real parameters to accomodate the general structure even if there are very few unique ones. It is necessary to use the most general structure to accomodate model averaging. Most of the time you will only want to compute the values of a limited set of real parameters but possibly for a range of covariate values. Use the argument indices to select the real parameters to be computed. The index is the value that the real parameter has been assigned with the all-different PIM structure. If you looked at the row numbers in the design data for the dipper example, you would see that the parameter for p and Phi are both numbered 1 to 21. But to deal with multiple parameters effectively they are given a unique number in a specific order. For the CJS model, p follows Phi, so for the dipper example, Phi are numbered 1 to 21 and then p are numbered 22 to 42. You can use the function PIMS to lookup the parameter numbers for a parameter type if you use simplified=FALSE. For example, with the dipper data, you could enter PIMS(dipper.model,"p",simplified=FALSE) and you would see that they are numbered 22 to 42. Alternatively, you can use summary.mark with the argument se=TRUE to see the list of indices named all.diff.index. They are included in a dataframe for each model parameter which enables them to be selected based on the attached data values (e.g., time, group etc). For example, if you fitted a model called dipper.model then you could use summary(dipper.model,se=TRUE)$real to list the indices for each parameter.

The argument data is a dataframe containing values for the covariates used in the models. The names for the fields should match the names of the covariates used in the model. If a time-varying

covariate is used then you need to specify the covariate name with the time index included as it is specified in the data. You do not need to specify all covariates that were used. If a covariate in one or more models is not included in `data` then the mean value will be used for each missing covariate. That can be very useful when you are only interested in prediction for one type of parameters (eg Phi) when there are many covariates that are not interesting in another parameter (e.g., p). For each row in `data`, each parameter specified in `indices` is computed with those covariate values. So if there were 5 rows in data and 10 parameters were specified there would be 10 sets of 5 (50) estimates produced. If you do not want the full pairing of data and estimates, create a field called `index` in `data` and the estimate for that parameter will be computed with those specific values. For example, if you wanted parameter 1 to be computed with 5 different values of a covariate and then parameter 7 with 2 different covariate values, you could create a dataframe with 5 rows each having an `index` with the value 1 along with the relevant covariate data and an additional 2 rows with the `index` with the value 7. If you include the field `index` in `data` then you do not need to give a value for the argument `indices`.

Value

A list is returned containing a dataframe of estimates, a var-cov matrix, and a reals list:

`estimates`	data frame containing estimates, se, confidence interval and the data values used to compute the estimates
`vcv`	variance-covariance matrix of real estimates
`reaks`	list of dataframes with the estimate and se used for each model

Author(s)

Jeff Laake

See Also

`compute.real`,`model.average`

Examples

```
pathtodata=paste(path.package("RMark"),"extdata",sep="/")

#
# indcov1.R
#
# CJS analysis of the individual covariate data from 12.2 of
# Cooch and White
#
# Import data (indcov1.inp) and convert it from the MARK inp file
# format to the RMark format using the function convert.inp
# It is defined with 1 group but 2 individual covariates of mass and
# squared.mass
#
indcov1=convert.inp(paste(pathtodata,"indcov1",sep="/"),
               covariates=c("mass","squared.mass"))
#
# Next create the processed dataframe and the design data.
```

```
#
  ind1.process=process.data(indcov1,model="CJS")
  ind1.ddl=make.design.data(ind1.process)
#
# Next create the function that defines and runs the set of models
# and returns a marklist with the results and a model.table.
# It does not have any arguments but does use the ind1.process
# and ind1.ddl objects created above in the workspace. The function
# create.model.list is the function that creates a dataframe of the
# names of the parameter specifications for each parameter in that
# type of model. If none are given for any parameter, the default
# specification will be used for that parameter in mark.  The
# first argument of mark.wrapper is the model list of parameter
# specifications.  Remaining arguments that are passed to
# mark must be specified using the argument=value specification
# because the arguments of mark were not repeated in mark.wrapper
# so they must be passed using the argument=value syntax.
#
ind1.models=function()
{
  Phi.dot=list(formula=~1)
  Phi.mass=list(formula=~mass)
  Phi.mass.plus.mass.squared=list(formula=~mass + squared.mass)
  p.dot=list(formula=~1)
  cml=create.model.list("CJS")
  results=mark.wrapper(cml,data=ind1.process,ddl=ind1.ddl,adjust=FALSE)
  return(results)
}
#
# Next run the function to create the models and store the results in
# ind1.results which is a marklist. Note that beta estimates will differ
# from Cooch and White results because we are using covariate values
# directly rather than standardized values.
#
ind1.results=ind1.models()
#
# Next compute real parameter values for survival as a function of
# mass which are model-averaged over the fitted models.
#
minmass=min(indcov1$mass)
maxmass=max(indcov1$mass)
mass.values=minmass+(0:30)*(maxmass-minmass)/30
Phibymass=covariate.predictions(ind1.results,
   data=data.frame(mass=mass.values,squared.mass=mass.values^2),
    indices=c(1))
#
# Plot predicted model averaged estimates by weight with pointwise
# confidence intervals
#
plot(Phibymass$estimates$mass, Phibymass$estimates$estimate,
   type="l",lwd=2,xlab="Mass(kg)",ylab="Survival",ylim=c(0,.65))
lines(Phibymass$estimates$mass, Phibymass$estimates$lcl,lty=2)
lines(Phibymass$estimates$mass, Phibymass$estimates$ucl,lty=2)
```

```
# indcov2.R
#
# CJS analysis of the individual covariate data from 12.3 of
# Cooch and White
#
# Import data (indcov2.inp) and convert it from the MARK inp file
# format to the RMark format using the function convert.inp  It is
# defined with 1 group but 2 individual covariates of mass and
# squared.mass
#
indcov2=convert.inp(paste(pathtodata,"indcov2",sep="/"),
        covariates=c("mass","squared.mass"))
#
# Standardize covariates
#
actual.mass=indcov2$mass
standardize=function(x,z=NULL)
{
   if(is.null(z))
   {
       return((x-mean(x))/sqrt(var(x)))
   }else
   {
      return((x-mean(z))/sqrt(var(z)))
   }
}
indcov2$mass=standardize(indcov2$mass)
indcov2$squared.mass=standardize(indcov2$squared.mass)
#
# Next create the processed dataframe and the design data.
#
  ind2.process=process.data(indcov2,model="CJS")
  ind2.ddl=make.design.data(ind2.process)
#
# Next create the function that defines and runs the set of models and
# returns a marklist with the results and a model.table.  It does not
# have any arguments but does use the ind1.process and ind1.ddl
# objects created above in the workspace. The function create.model.list
# is the function that creates a dataframe of the names of the parameter
# specifications for each parameter in that type of model. If none are
# given for any parameter, the default specification will be used for
# that parameter in mark.  The first argument of mark.wrapper is the
# model list of parameter specifications.  Remaining arguments that are
# passed to mark must be specified using the argument=value specification
# because the arguments of mark were not repeated in mark.wrapper so
# they must be passed using the argument=value syntax.
#
ind2.models=function()
{
  Phi.dot=list(formula=~1)
  Phi.time=list(formula=~time)
  Phi.mass=list(formula=~mass)
```

```
  Phi.mass.plus.mass.squared=list(formula=~mass + squared.mass)
  Phi.time.x.mass.plus.mass.squared=
        list(formula=~time:mass + time:squared.mass)
  Phi.time.mass.plus.mass.squared=
     list(formula=~time*mass + squared.mass+ time:squared.mass)
  p.dot=list(formula=~1)
  cml=create.model.list("CJS")
  results=mark.wrapper(cml,data=ind2.process,ddl=ind2.ddl,adjust=FALSE,threads=2)
  return(results)
}
#
# Next run the function to create the models and store the results in
# ind2.results which is a marklist. Note that beta estimates will differ
# because we are using covariate values directly rather than
# standardized values.
#
ind2.results=ind2.models()
#
# Next compute real parameter values for survival as a function of
# mass which are model-averaged over the fitted models. They are
# standardized individually so the values have to be chosen differently.
#
minmass=min(actual.mass)
maxmass=max(actual.mass)
mass.values=minmass+(0:30)*(maxmass-minmass)/30
squared.mass.values=mass.values^2
mass.values=standardize(mass.values,actual.mass)
squared.mass.values=standardize(squared.mass.values,actual.mass^2)
Phibymass=covariate.predictions(ind2.results,
 data=data.frame(mass=mass.values,squared.mass=squared.mass.values),
  indices=c(1:7))
#
# Plot predicted model averaged estimates by weight with pointwise
# confidence intervals
#
par(mfrow=c(4,2))
for (i in 1:7)
{
   mass=minmass+(0:30)*(maxmass-minmass)/30
   x=Phibymass$estimates
   plot(mass,x$estimate[x$par.index==i],type="l",lwd=2,
    xlab="Mass(kg)",ylab="Survival",ylim=c(0,1),main=paste("Time",i))
   lines(mass, x$lcl[x$par.index==i],lty=2)
   lines(mass, x$ucl[x$par.index==i],lty=2)
}
```

---

crdms *Example data for Closed Robust Design Multistrata*

Description

Data and Script to simulate the MSCRD example of 15.7.1 from the MARK book Cooch and White

Format

A data frame with 557 observations on the following 2 variables.

ch a character vector of encounter histories

freq a numeric vector of frequencies of each history

Source

This example was constructed by Andrew Paul who is with Fish and Wildlife Division of the Alberta Provincial Government

References

For Cooch and White book see http://www.phidot.org/software/mark/

Examples

```
#Script to simulate the MSCRD
#example of 15.7.1 from the MARK
#book
#created by AJP 21 Dec 2010

#cleanup the R environment
graphics.off()
rm(list=ls())

#cleanup files
cleanup(ask=FALSE)

#convert .inp data - only needed to create crdms
#ch.data<-convert.inp("rd_simple1.inp")
data(crdms)
#set time intervals
#4 primary periods each with 3 secondary occasions
t.int<-c(rep(c(0,0,1),3),c(0,0))

#process data for RMark
crdms.data<-process.data(crdms,model="CRDMS",time.interval=t.int,
strata.labels=c("1","U"))
#change Psi parameters that are obtained by subtraction
crdms.ddl<-make.design.data(crdms.data,
parameters=list(Psi=list(subtract.stratum=c("1","1"))))

#create grouping index for unobserved p and c (i.e., always zero)
up=as.numeric(row.names(crdms.ddl$p[crdms.ddl$p$stratum=="U",]))
```

```
#create grouping index to fix Psi for unobs to unbos at time 1
#this isn't necessary but it allows this Psi to be fixed to a value
#that can be flagged and not erroneously interpreted
Psiuu1=as.numeric(row.names(crdms.ddl$Psi[crdms.ddl$Psi$stratum=="U"&
crdms.ddl$Psi$time==1,]))

#create dummy variable for constraining last Psi in Markovian model
#variable is called ctime for constrained time
crdms.ddl$Psi$ctime=crdms.ddl$Psi$time
crdms.ddl$Psi$ctime[crdms.ddl$Psi$time==3]=2

#Initial assumptions
S.dot=list(formula=~1)  #S equal for both states and constant over time
p.session=list(formula=~session, share=TRUE,  #p=c varies with session
fixed=list(index=up,value=0)) #p set to zero for unobs

#Model 1 - Markovian movement
Psi.markov=list(formula=~ctime+stratum,
fixed=list(index=Psiuu1,value=9e-99)) #9e-99 is a flag
model.1=mark(crdms.data,ddl=crdms.ddl,
model.parameters=list(S=S.dot,
p=p.session,
Psi=Psi.markov),threads=2)

#Model 2 - Random movement
Psi.rand=list(formula=~time)
model.2=mark(crdms.data,ddl=crdms.ddl,
model.parameters=list(S=S.dot,
p=p.session,
Psi=Psi.rand),threads=2)

#Model 3 - No movement
Psi.fix=list(formula=~1,fixed=0)
model.3=mark(crdms.data,ddl=crdms.ddl,
model.parameters=list(S=S.dot,
p=p.session,
Psi=Psi.fix),threads=2)

#collect and store models
crdms.res<-collect.models()

print(crdms.res)

#final cleanup
cleanup(ask=FALSE)
```

---

create.mark.mcmc *Create mcmc object for analysis with coda*

Description

Reads in mcmc file from program MARK in binary and returns an mcmc object that can be used with coda functions which are most easily accessed via codamenu().

Usage

```
create.mark.mcmc(filename, ncovs, nmeans, ndesigns, nsigmas, nrhos, nlogit,
  include = F)
```

Arguments

filename	name of file containing mcmc values
ncovs	number of covariates
nmeans	number of means
ndesigns	number of designs
nsigmas	number of sigmas
nrhos	number of rhos
nlogit	number of logits
include	if TRUE it includes ir/propJumps fields

Value

An mcmc object if one chain and an mcmc list if more than one chain. Each can be used with the coda package

Author(s)

Jeff Laake

---

create.model.list *Creates a dataframe of all combinations of parameter specifications*

Description

Creates a dataframe of all combinations of parameter specifications for each parameter in a particular type of MARK model. It is used together with mark.wrapper to run a series of models from sets of parameter specifications.

Usage

```
create.model.list(model)
```

Arguments

model	character string identifying the type of model (e.g., "CJS")

Details

This function scans the frame of the calling enviroment and collects all list objects that contain a formula and have names that match `parameter.` where parameter is the name of a type of parameter in the `model` type. For example, it looks for `Phi.` and `p.` for `model="CJS"`. Any number of characters can follow the period. Each of the named objects should specify a list that matches the structure of a parameter specification as described in `make.mark.model`. It only collects list objects that contain an element named `formula`, thus it will not collect one like `Phi.fixed=list(fixed=1)`. If you want to do something like that, specify it as `Phi.fixed=list(formula=~1,fixed=1)`. It is safest to use this inside a function that defines all of the parameter specifications as shown in the example below. The primary use for this function is to create a dataframe which is passed to `mark.wrapper` to construct and run each of the models. It was written as a separate function to provide flexibility to add/delete/modify the list prior to passing to `mark.wrapper`. For example, only certain combinations may make sense for some parameter specifications. Thus you could define a set to create all the combinations and then delete the ones from the dataframe that do not make sense. you want, add others and re-run the function and merge the resulting dataframes. If there are no specifications found for a particular model parameter, it is not included in the list and when it is passed to `make.mark.model`, the default specification will be used for that parameter.

Value

dataframe of all combinations of parameter specifications for a model. Each field (column) is the name of a type of parameter (e.g., p and Phi for CJS). The values are character strings identifying particular parameter specifications.

Author(s)

Jeff Laake

See Also

`mark.wrapper`

Examples

```
#
# Compare this to the run.dipper shown under ?dipper
# It is only necessary to create each parameter specification and
# create.model.list and mark.wrapper will create and run models for
# each combination. Notice that the naming of the parameter
# specifications has been changed to accommodate format for
# create.model.list. Only a subset of the parameter specifications
# are used here in comparison to other run.dipper
#
data(dipper)
run.dipper=function()
{
     #
     # Process data
     #
```

```
  dipper.processed=process.data(dipper,groups=("sex"))
  #
  # Create default design data
  #
  dipper.ddl=make.design.data(dipper.processed)
  #
  # Add Flood covariates for Phi and p that have different values
  #
  dipper.ddl$Phi$Flood=0
  dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 |
               dipper.ddl$Phi$time==3]=1
  dipper.ddl$p$Flood=0
  dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
  #
  #  Define range of models for Phi
  #
  Phi.dot=list(formula=~1)
  Phi.time=list(formula=~time)
  Phi.sex=list(formula=~sex)
  Phi.Flood=list(formula=~Flood)
  #
  #  Define range of models for p
  #
  p.dot=list(formula=~1)
  p.time=list(formula=~time)
  p.Flood=list(formula=~Flood)
  #
  # Run all pairings of models
  #
  dipper.model.list=create.model.list("CJS")
  dipper.results=mark.wrapper(dipper.model.list,
           data=dipper.processed,ddl=dipper.ddl)
  #
  # Return model table and list of models
  #
  return(dipper.results)
}
dipper.results=run.dipper()
```

---

deer *White-tailed deer double observer spotlight capture-recapture analysis*

Description

This data represents a set of independent double observer road-transect survey data of white-tailed deer on Brosnan Forest, South Carolina surveyed in August, 2005-2009. The primary reason for this package is to provide a completely reproducible example of the analysis from Collier et al. (2012). We used a Huggins closed capture model implemented in MARK http://www.phidot.

org/software/mark/ via RMark http://cran.r-project.org/web/packages/RMark/index.html both of which will need to be installed on the system to use this package. The data have 2 time periods (primary observer (t1) was a thermal imager, secondary observer (t2) was a spotlight observer in the same vehicle on the same side) with the primary objective of the study being to evaluate the detection (recapture) rates of white-tailed deer using spotlights as a survey method.

Format

The format is a data frame with 4508 observations on the following 7 variables.

SL (spotlight) 0/1 whether deer was missed/seen by the spotlight observer

TI (thermal imager) 0/1 whether deer was missed/seen by the thermal imager observer

Group Factor with 79 levels representing each unique paired (TI-SL) survey conducted

Year Factor with 5 levels for year of survey

MaxCount Count of maximum number of deer seen for each survey, only needed for bootstrapp analysis in MARK, not used in bfdeeR package

Cluster Value assigning each deer to a specific observation cluster, only needed for bootstrapp analysis in MARK, not used in bfdeeR package

MgmtUnit Management unit identification

Details

In addition to detailing the analysis used by Collier et al. (2012), this example documents the use of the `share` argument in the RMark parameter specification because there is presently very little documentation on the use of `share`. Parameters in MARK models rarely share columns of the design matrix. For example while you might want to use the same covariate for survival and capture probability, you would never use the same beta (same column of the design matrix) for each parameter. However, there are exceptions when the parameters represent similar quantities and that is when the `share` argument is useful. For example, in the closed capture models p is initial capture probability and c is recapture probability. In this case, it would make perfect sense to use the same column of the design matrix for both parameters. The most obvious case is to fit a model in which p=c.

In RMark, certain pairs of parameters have been identified as similar and shareable. These can be found in the file parameters.txt which is in the RMark directory in your R library. With each pair that is shareable, the first one listed is the primary parameter. When you want to share columns in the design matrix, share=TRUE is added to the specification of the primary parameter. A parameter specification is not given for the other secondary parameter when they are shared. When RMark, sees that the parameters are to be shared it creates a pooled set of design data and adds a column with the name of the secondary parameter and its value is 0 for the rows for the primary parameter and 1 for the rows for the secondary parameter. For example, with the closed capture model if share=TRUE is added to the parameter specification for p, a model is not specified for c, and the pooled design data set contains a field called c. The added field allows construction of models where there are restricted differences between the parameters. For example, p=list(formula=~time+c,share=TRUE) will fit a model in which capture probability varies by time and recapture probability includes an additive difference on the link scale. Because the design data are pooled when you share parameters, if you modify design data for one of the parameters, the other most be modified as as well, so the columns of the design data for both parameters are the same or RMark will give an error.

The argument share is used in all the candidate models in the below example analysis. As a simplified example of how share works, look at the candidate models in the bfrun{} function call named mod.2 and mod.2a (note that mod.2a was not included in the supplemental file available from the Journal of Wildlife Management and is only included in this package). Both of these models are conducting the exact same analysis, with the first mod.2, we used the formula ~time (if you don't know what this means go read the MARKBOOK at http://www.phidot.org/software/mark/. Notice, however, we used the argument share in mod.2, which tells RMark to share columns of the MARK design matrix. For comparison, so you can evaluate how share works for yourself, mod.2a recreates the same analysis as mod.2, but uses the approach more typical to MARK analyses where each parameter is specified independently and uniquely.

Author(s)

Bret Collier

References

Collier, B. A., S. S. Ditchkoff, J. B. Raglin, and C. R. Ruth. 2012. Spotlight surveys for white-tailed deer: monitoring panacea or exercise in futility? Journal of Wildlife Management, In Press.

Examples

```
data(deer)
x=data.frame(ch=paste(deer$TI, deer$SL, sep=""), Survey=factor(deer$Group),
     Year=factor(deer$Year), Cluster=deer$Cluster, MgtUnit=factor(deer$MgmtUnit))
x$ch=as.character(x$ch)
bfrun=function(){
x.proc=process.data(x, model="Huggins", groups=c("Survey", "Year", "MgtUnit"))
x.ddl=make.design.data(x.proc)

#Silly Null model, constant p & c sharing 1 parameter (one detection estimate)
p.shared=list(formula=~1,share=TRUE)
mod.1=mark(x.proc, x.ddl, model.parameters=list(p=p.shared), invisible=FALSE)

#2 Parameter Null Model, constant p, constant c, different p and c (one estimate for each; p ne c)
#p(time), c(-), share=TRUE, detection is time dependent, with recapture parameter shared
p.sharetime=list(formula=~time, share=TRUE)
mod.2=mark(x.proc, x.ddl, model.parameters=list(p=p.sharetime), invisible=FALSE)

#2a Parameter Null Model, constant p, constant c,
# different p and c (one estimate for each; p ne c) not using share
mod.2a=mark(x.proc, x.ddl, model.parameters=list(p=list(formula=~1), c=list(formula=~1)))

#Fully parameterized model, different p and c for each survey transect replicate,
# management unit, method (TI or SL) and any observers
p.survey=list(formula=~Survey*time, share=TRUE)
mod.3=mark(x.proc, x.ddl, model.parameters=list(p=p.survey), invisible=FALSE)

#p(MU), c(MU), initial detection and recapture differ and are management unit dependent
p.mu=list(formula=~MgtUnit*time, share=TRUE)
mod.4=mark(x.proc, x.ddl, model.parameters=list(p=p.mu), invisible=FALSE)
```

```
#p(MU) detection is management unit dependent
p.mu=list(formula=~MgtUnit, share=TRUE)
mod.5=mark(x.proc, x.ddl, model.parameters=list(p=p.mu), invisible=FALSE)

#p(Yr + MgtUnit),  detection is year + MgtUnit
p.yearMgtUnit=list(formula=~Year*time+MgtUnit, share=TRUE)
mod.6=mark(x.proc, x.ddl, model.parameters=list(p=p.yearMgtUnit), invisible=FALSE)

#p(Year), initial detection and recapture are year dependent
p.year=list(formula=~Year*time, share=TRUE)
mod.7=mark(x.proc, x.ddl, model.parameters=list(p=p.year), invisible=FALSE)

return(collect.models())
}
bf.out=bfrun()
bf.out

#export function to send dataset and covariates data to MARK for bootstrap analysis
#(not run but here for completeness)
#export.MARK(x.proc, "BFdeer", mod.3, replace=TRUE, ind.covariates="all")
```

`deltamethod.special` *Compute delta method variance for sum, cumsum, prod and cumprod functions*

Description

This function computes the delta method std errors or v-c matrix for a sum, cumsum (vector of cummulative sums), prod (product), or cumprod(vector of cummulative products) of a set of estimates.

Usage

```
deltamethod.special(function.name, mean, cov, ses = TRUE)
```

Arguments

`function.name`	Quoted character string of either "sum", "cumsum", "prod" or "cumprod"
`mean`	vector of estimates used in the function
`cov`	variance-covariance matrix of the estimates
`ses`	if TRUE it returns a vector of estimated standard errors of the function of the estimates and if FALSE it returns the variance-covariance matrix

Details

This function computes the delta method std errors or v-c matrix for a sum, cumsum (vector of cummulative sums), prod (product), or cumprod(vector of cummulative products). It uses the function deltamethod from the msm package and constructs the necessary formula for these special cases. It will load the msm pacakge but assumes that it has already been installed. See the msm documentation for a complete description on how the deltamethod function works. If ses=TRUE, it returns a vector of std errors for each of functions of the estimates contained in mean. If ses=F, then it returns a v-c matrix for the functions of the estimates contained in mean. cov is the input v-c matrix of the estimates.

Value

either a vector of standard errors (ses=TRUE) or a variance-covariance matrix (ses=FALSE)

Author(s)

Jeff Laake

Examples

```
#
# The following are examples only to demonstrate selecting different
# model sets for adjusting chat and showing model selection table.
# It is not a realistic analysis.
#
  data(dipper)
  mod=mark(dipper,model.parameters=list(Phi=list(formula=~time)))
  rr=get.real(mod,"Phi",se=TRUE,vcv=TRUE)
  deltamethod.special("prod",rr$estimates$estimate[1:6],rr$vcv.real)
  deltamethod.special("cumprod",rr$estimates$estimate[1:6],rr$vcv.real,ses=FALSE)
  deltamethod.special("sum",rr$estimates$estimate[1:6],rr$vcv.real)
  deltamethod.special("cumsum",rr$estimates$estimate[1:6],rr$vcv.real,ses=FALSE)
```

---

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. See `adjust.parameter.count` for more dicussion on this point.

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
```

See `export.chdata` for an example that creates a .inp file for MARK with sex being used to describe groups and a numeric sex covariate.

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.

Examples

```
data(dipper)
dipper.model=mark(dipper)
run.dipper=function()
{
#
# Process data
#
dipper.processed=process.data(dipper,groups=("sex"))
#
# Create default design data
```

```
#
dipper.ddl=make.design.data(dipper.processed)
#
# Add Flood covariates for Phi and p that have different values
#
dipper.ddl$Phi$Flood=0
dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 | dipper.ddl$Phi$time==3]=1
dipper.ddl$p$Flood=0
dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
#
#  Define range of models for Phi
#
Phidot=list(formula=~1)
Phitime=list(formula=~time)
Phisex=list(formula=~sex)
Phisextime=list(formula=~sex+time)
Phisex.time=list(formula=~sex*time)
PhiFlood=list(formula=~Flood)
#
#  Define range of models for p
#
pdot=list(formula=~1)
ptime=list(formula=~time)
psex=list(formula=~sex)
psextime=list(formula=~sex+time)
psex.time=list(formula=~sex*time)
pFlood=list(formula=~Flood)
#
# Run assortment of models
#
dipper.phidot.pdot          =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phidot,p=pdot))
dipper.phidot.pFlood       =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phidot,p=pFlood))
dipper.phidot.psex         =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phidot,p=psex))
dipper.phidot.ptime        =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phidot,p=ptime))
dipper.phidot.psex.time =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phidot,p=psex.time))
dipper.phitime.ptime       =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phitime, p=ptime))
dipper.phitime.pdot        =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phitime,p=pdot))
dipper.phitime.psex =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phitime,p=psex))
dipper.phitime.psex.time =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phitime,p=psex.time))
dipper.phiFlood.pFlood     =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=PhiFlood, p=pFlood))
dipper.phisex.pdot         =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex,p=pdot))
dipper.phisex.psex         =mark(dipper.processed,dipper.ddl,
```

```
                 model.parameters=list(Phi=Phisex,p=psex))
dipper.phisex.psex.time         =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex,p=psex.time))
dipper.phisex.ptime        =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex,p=ptime))
dipper.phisextime.psextime =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisextime,p=psextime))
dipper.phisex.time.psex.time =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex.time,p=psex.time))
dipper.phisex.time.psex  =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex.time,p=psex))
dipper.phisex.time.pdot =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex.time,p=pdot))
dipper.phisex.time.ptime =mark(dipper.processed,dipper.ddl,
                 model.parameters=list(Phi=Phisex.time,p=ptime))
#
# Return model table and list of models
#
return(collect.models() )
}

dipper.results=run.dipper()

run.dipper.alternate=function()
{
#
# Process data
#
dipper.processed=process.data(dipper,groups=("sex"))
#
# Create default design data
#
dipper.ddl=make.design.data(dipper.processed)
#
# Add Flood covariates for Phi and p that have different values
#
dipper.ddl$Phi$Flood=0
dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 | dipper.ddl$Phi$time==3]=1
dipper.ddl$p$Flood=0
dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
#
#  Define range of models for Phi
#
Phidot=list(formula=~1)
Phitime=list(formula=~time)
Phitimec=list(formula=~time,fixed=list(time=6,value=1))
Phisex=list(formula=~sex)
Phisextime=list(formula=~sex+time)
Phisex.time=list(formula=~sex*time)
PhiFlood=list(formula=~Flood)
#
#  Define range of models for p
#
```

```
pdot=list(formula=~1)
ptime=list(formula=~time)
ptimec=list(formula=~time,fixed=list(time=7,value=1))
psex=list(formula=~sex)
psextime=list(formula=~sex+time)
psex.time=list(formula=~sex*time)
psex.timec=list(formula=~sex*time,fixed=list(time=7,value=1))
pFlood=list(formula=~Flood)
#
# Run assortment of models
#
dipper.phidot.pdot          =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phidot,p=pdot))
dipper.phidot.pFlood       =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phidot,p=pFlood))
dipper.phidot.psex         =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phidot,p=psex))
dipper.phidot.ptime        =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phidot,p=ptime))
dipper.phidot.psex.time =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phidot,p=psex.time))
dipper.phitime.ptimec       =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phitime, p=ptimec))
dipper.phitime.pdot        =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phitime,p=pdot))
dipper.phitime.psex =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phitime,p=psex))
dipper.phitimec.psex.time =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phitimec,p=psex.time))
dipper.phiFlood.pFlood     =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=PhiFlood, p=pFlood))
dipper.phisex.pdot         =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex,p=pdot))
dipper.phisex.psex         =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex,p=psex))
dipper.phisex.psex.time         =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex,p=psex.time))
dipper.phisex.ptime        =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex,p=ptime))
dipper.phisextime.psextime =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisextime,p=psextime),adjust=FALSE)
dipper.phisex.time.psex.timec =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex.time,p=psex.timec))
dipper.phisex.time.psex  =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex.time,p=psex))
dipper.phisex.time.pdot =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex.time,p=pdot))
dipper.phisex.time.ptimec =mark(dipper.processed,dipper.ddl,
                  model.parameters=list(Phi=Phisex.time,p=ptimec))
#
# Return model table and list of models
#
return(collect.models() )
```

```
}
dipper.results.alternate=run.dipper.alternate()
#
# Merge two sets of models into a single model list and include the
# initial model as a demo for merge.mark
#
dipper.cjs=merge.mark(dipper.results,dipper.results.alternate,dipper.model)
dipper.cjs
#
# next delete some of the models to show how this is done with remove.mark
#
dipper.cjs=remove.mark(dipper.cjs,c(2,4,9))
dipper.cjs

run.dipper.popan=function()
{
#
# Process data
#
dipper.processed=process.data(dipper,model="POPAN",group="sex")
#
# Create default design data
#
dipper.ddl=make.design.data(dipper.processed)
#
# Add Flood covariates for Phi and p that have different values
#
dipper.ddl$Phi$Flood=0
dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 | dipper.ddl$Phi$time==3]=1
dipper.ddl$p$Flood=0
dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
#
#  Define range of models for Phi
#
Phidot=list(formula=~1)
Phitime=list(formula=~time)
Phisex=list(formula=~sex)
Phisextime=list(formula=~sex+time)
Phisex.time=list(formula=~sex*time)
PhiFlood=list(formula=~Flood)
#
#  Define range of models for p
#
pdot=list(formula=~1)
ptime=list(formula=~time)
psex=list(formula=~sex)
psextime=list(formula=~sex+time)
psex.time=list(formula=~sex*time)
pFlood=list(formula=~Flood)
#
#  Define range of models for pent
#
pentsex.time=list(formula=~sex*time)
```

```
#
#  Define range of models for N
#
Nsex=list(formula=~sex)
#
# Run assortment of models
#
dipper.phisex.time.psex.time.pentsex.time=mark(dipper.processed,dipper.ddl,
model.parameters=list(Phi=Phisex.time,p=psex.time,pent=pentsex.time,N=Nsex),
invisible=FALSE,adjust=FALSE)
dipper.phisex.time.psex.pentsex.time=mark(dipper.processed,dipper.ddl,
model.parameters=list(Phi=Phisex.time,p=psex,pent=pentsex.time,N=Nsex),
invisible=FALSE,adjust=FALSE)
#
# Return model table and list of models
#
return(collect.models() )
}

dipper.popan.results=run.dipper.popan()

# ****************************************************************
# Here is an example of user specified links for each real parameter
 data(dipper)
 dipper.proc=process.data(dipper)
 dipper.ddl=make.design.data(dipper.proc)
# dummy run of make.mark.model to get links and design data.
# parm.specific set to TRUE so it will create a link for
# each parameter because for this model they are all the
# same (logit) and if this was not specified you'ld get a vector with one element
 dummy=make.mark.model(dipper.proc,dipper.ddl,simplify=FALSE,parm.specific=TRUE)
 input.links=dummy$links
# get model indices for p where time=4
 log.indices=dipper.ddl$p$model.index[dipper.ddl$p$time==4]
# assign those links to log
 input.links[log.indices]="Log"
# Now these can be used with any call to mark
 mymodel=mark(dipper.proc,dipper.ddl,input.links=input.links)
 summary(mymodel)
```

Donovan.7 *Exercise 7 example data*

Description

An example occupancy data set used as exercise 7 in the occupancy website developed by Donovan and Hines.

Format

A data frame with 20 observations (sites) on the following 2 variables.

ch a character vector containing the presence (1) and absence (0) for each of 5 visits to the site

freq frequency of sites (always 1)

Details

This is a data set from exercise 7 of Donovan and Hines occupancy web site (http://www.uvm.edu/envnr/vtcfwru/spreadsheets/occupancy/occupancy.htm).

Examples

```
# Donovan.7 can be created with
# Donovan.7=convert.inp("Donovan.7.inp")

do.exercise.7=function()
{
  data(Donovan.7)
# Estimates from following agree with estimates on website but the
# log-likelihood values do not agree.  Maybe a difference in whether the
# constant binomial coefficients are included.
  Donovan.7.poisson=mark(Donovan.7,model="OccupRNPoisson",invisible=FALSE,threads=2)
# THe following model was not in exercise 7.
  Donovan.7.negbin=mark(Donovan.7,model="OccupRNNegBin",invisible=FALSE,threads=2)
  return(collect.models())
}
exercise.7=do.exercise.7()
# Remove # to see output
# print(exercise.7)
```

Donovan.8 *Exercise 8 example data*

Description

An example occupancy data set used as exercise 8 in the occupancy website developed by Donovan and Hines.

Format

A data frame with 20 observations (sites) on the following 2 variables.

ch a character vector containing the counts for each of 5 visits to the site

freq frequency of sites (always 1)

Details

This is a data set from exercise 8 of Donovan and Hines occupancy web site (http://www.uvm.edu/envnr/vtcfwru/spreadsheets/occupancy/occupancy.htm). In MARK, it uses 2 digits to allow a count of 0 to 99 at each site, so the history has 10 digits for 5 visits (occasions).

Examples

```
# Donovan.8 can be created with
# Donovan.8=convert.inp("Donovan.8.inp")
do.exercise.8=function()
{
  data(Donovan.8)
# Results agree with the values on the website.
  Donovan.8.poisson=mark(Donovan.8,model="OccupRPoisson",invisible=FALSE,threads=2)
# The following model was not in exercise 8. The NegBin model does
# better if it is initialized with the r and lambda from the poisson.
  Donovan.8.negbin=mark(Donovan.8,model="OccupRNegBin",
    initial=Donovan.8.poisson,invisible=FALSE,threads=2)
  return(collect.models())
}
exercise.8=do.exercise.8()
# Remove # to see output
# print(exercise.8)
```

---

edwards.eberhardt *Rabbit capture-recapture data*

Description

A capture-recapture data set on rabbits derived from Edwards and Eberhardt (1967) that accompanies MARK as an example analysis using the closed population models.

Format

A data frame with 76 observations on the following variable.

ch a character vector

Details

This data set is used in MARK to illustrate the various closed population models including "Closed", "HetClosed", "FullHet","Huggins","HugHet", and "FullHugHet". The first 3 include N in the likelihood whereas the last 3 are based on the Huggins approach which does not use N in the likelihood. The Het... and FullHet... models are based on the Pledger mixture model approach. Some of the examples demonstrate the use of the `share` argument in the `model.parameters` list for parameter `p` which allows sharing common values for `p` and `c`.

Source

Edwards, W.R. and L.L. Eberhardt 1967. Estimating cottontail abundance from live trapping data. J. Wildl. Manage. 31:87-96.

Examples

```
#
# get data
#
data(edwards.eberhardt)
#
# create function that defines and runs the analyses as defined in
# MARK example dbf file
#
run.edwards.eberhardt=function()
{
#
#  Define parameter models
#
pdotshared=list(formula=~1,share=TRUE)
ptimeshared=list(formula=~time,share=TRUE)
ptime.c=list(formula=~time+c,share=TRUE)
ptimemixtureshared=list(formula=~time+mixture,share=TRUE)
pmixture=list(formula=~mixture)
#
# Run assortment of models
#
#
#   Capture Closed models
#
#  constant p=c
ee.closed.m0=mark(edwards.eberhardt,model="Closed",
                  model.parameters=list(p=pdotshared))
#  constant p and constant c but different
ee.closed.m0c=mark(edwards.eberhardt,model="Closed")
#  time varying p=c
ee.closed.mt=mark(edwards.eberhardt,model="Closed",
                  model.parameters=list(p=ptimeshared))
#
#  Closed heterogeneity models
#
#  2 mixtures Mh2
ee.closed.Mh2=mark(edwards.eberhardt,model="HetClosed",
                  model.parameters=list(p=pmixture))
#  Closed Mth2 - p different for time; mixture additive
ee.closed.Mth2.additive=mark(edwards.eberhardt,model="FullHet",
                  model.parameters=list(p=ptimemixtureshared),adjust=TRUE)
#
#    Huggins models
#
# p=c constant over time
```

```
ee.huggins.m0=mark(edwards.eberhardt,model="Huggins",
                   model.parameters=list(p=pdotshared))
# p constant c constant but different; this is default model for Huggins
ee.huggins.m0.c=mark(edwards.eberhardt,model="Huggins")
# Huggins Mt
ee.huggins.Mt=mark(edwards.eberhardt,model="Huggins",
                   model.parameters=list(p=ptimeshared),adjust=TRUE)
#
#    Huggins heterogeneity models
#
#  Mh2 - p different for mixture
ee.huggins.Mh2=mark(edwards.eberhardt,model="HugHet",
                   model.parameters=list(p=pmixture))
#  Huggins Mth2 - p different for time; mixture additive
ee.huggins.Mth2.additive=mark(edwards.eberhardt,model="HugFullHet",
                   model.parameters=list(p=ptimemixtureshared),adjust=TRUE)
#
# Return model table and list of models
#
return(collect.models() )
}
#
# fit models in mark by calling function created above
#
ee.results=run.edwards.eberhardt()
```

`example.data` *Simulated data from Cormack-Jolly-Seber model*

Description

A simulated data set from CJS model to demonstrate use of grouping variables and individual covariates in an analysis of a `mark` model. The true model for the simulated data is S(age+year)p(year).

Format

A data frame with 6000 observations on the following 5 variables.

ch a character vector

weight a numeric vector

age a factor with levels `1` `2` `3`

sex a factor with levels `F` `M`

region a factor with levels `1` `2` `3` `4`

Details

The `weight`, `age` and `region` are static variables that are defined based on the values when the animal was released. `age` is a factor variable representing an age level. The actual ages (at time of release) are 0,1,2 for the 3 levels respectively.

Examples

```
data(example.data)
run.example=function()
{
PhiTime=list(formula=~time)
pTimec=list(formula=~time,fixed=list(time=7,value=1))
pTime=list(formula=~time)
PhiAge=list(formula=~age)
Phidot=list(formula=~1)
PhiweightTime=list(formula=~weight+time)
PhiTimeAge=list(formula=~time+age)
mod1=mark(example.data,groups=c("sex","age","region"),
                           initial.ages=c(0,1,2))
mod2=mark(example.data,model.parameters=list(p=pTimec,Phi=PhiTime),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
mod3=mark(example.data,model.parameters=list(Phi=Phidot,p=pTime),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
mod4=mark(example.data,model.parameters=list(Phi=PhiTime),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
mod5=mark(example.data,model.parameters=list(Phi=PhiTimeAge),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
mod6=mark(example.data,model.parameters=list(Phi=PhiAge,p=pTime),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
mod7=mark(example.data,model.parameters=list(p=pTime,Phi=PhiweightTime),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
mod8=mark(example.data,model.parameters=list(Phi=PhiTimeAge,p=pTime),
          groups=c("sex","age","region"),initial.ages=c(0,1,2))
return(collect.models())
}
example.results=run.example()
```

`export.chdata` *Export capture-history data to MARK .inp format*

Description

Creates a MARK .inp file from processed data list that can be used to create a MARK .dbf file for use with MARK directly rather than with the RMark package.

Usage

```
export.chdata(data, filename, covariates = NULL, replace = FALSE)
```

Arguments

`data`	processed data list resulting from process.data
`filename`	quoted filename (without .inp extension)

covariates	vector of names of covariate variables in data to include
replace	if file exists and replace=TRUE, file will be over-written

Details

The default is to include none of the covariates in the processed data list. All of the covariates can be passed by setting covariates="all".

After you have created the MARK .dbf/.fpt files with the exported .inp file, then you can use export.chdata to export models that can be imported into the MARK interface. However note the following: ***Warning*** Make sure that you use the .inp created by export.chdata with your processed data to create the MARK .dbf file rather than using a separate similar .inp file. It is essential that the group structure and ordering of groups matches between the .inp file and the exported models or you can get erroneous results.

Value

None

Author(s)

Jeff Laake

See Also

import.chdata

Examples

```
data(dipper)
dipper$numeric.sex=as.numeric(dipper$sex)-1
dipper.processed=process.data(dipper,group="sex")
export.chdata(dipper.processed, filename="dipper",
         covariates="numeric.sex",replace=TRUE)
#
# Had sex been used in place of numeric.sex in the above command,
# MARK would have been unable to use it as a covariate
# because it is not a numeric field
```

---

export.MARK *Export data and models for import in MARK*

Description

Creates a .Rinp, .inp and optionally renamed output files that can be imported into MARK to create a new MARK project with all the data and output files.

Usage

```
export.MARK(x, project.name, model = NULL, replace = FALSE, chat = 1,
  title = "", ind.covariates = "all")
```

Arguments

x	processed data list used to build models
project.name	character string to be used for prefix of filenames and for MARK project name; do not use "." in the filename
model	either a single mark model or a marklist
replace	if TRUE it will replace any existing files
chat	user-specified chat value if desired
title	MARK project title string
ind.covariates	vector of character strings specifying names of individual covariates or "all" to use all present

Details

If you use Nest model and NestAge covariate you should use the processed data list (model$data) from a model using NestAge in the formula because the necessary individual covariates are added to the processed data list. Also, use default of ind.covariates="all".

After running this function to export the data and models to the working directory, start program MARK and select the File/RMARK Import menu item. Navigate to the working directory and select the "project.name".Rinp file. MARK will take over and create the project files and will import any specified model output files.

DO NOT use a "project name" that is the same as a filename you use as input (*.inp) to RMark because this function will create a file "project name".inp and it would over-write your existing file and the format could change. If you try this, the code will return an error that the filename is invalid because the file already exists.

Value

None

Author(s)

Jeff Laake

See Also

export.chdata, export.model

Examples

```
data(mstrata)
run.mstrata=function()
{
#
# Process data
#
mstrata.processed=process.data(mstrata,model="Multistrata")
#
# Create default design data
#
mstrata.ddl=make.design.data(mstrata.processed)
#
#  Define range of models for S; note that the betas will differ from the output
#  in MARK for the ~stratum = S(s) because the design matrix is defined using
#  treatment contrasts for factors so the intercept is stratum A and the other
#  two estimates represent the amount that survival for B abd C differ from A.
#  You can use force the approach used in MARK with the formula ~-1+stratum which
#  creates 3 separate Betas - one for A,B and C.
#
S.stratum=list(formula=~stratum)
S.stratumxtime=list(formula=~stratum*time)
#
#  Define range of models for p
#
p.stratum=list(formula=~stratum)
#
#  Define range of models for Psi; what is denoted as s for Psi
#  in the Mark example for Psi is accomplished by -1+stratum:tostratum which
#  nests tostratum within stratum.  Likewise, to get s*t as noted in MARK you
#  want ~-1+stratum:tostratum:time with time nested in tostratum nested in
#  stratum.
#
Psi.s=list(formula=~-1+stratum:tostratum)
Psi.sxtime=list(formula=~-1+stratum:tostratum:time)
#
# Create model list and run assortment of models
#
model.list=create.model.list("Multistrata")
#
# Add on specific models that are paired with fixed p's to remove confounding
#
p.stratumxtime=list(formula=~stratum*time)
p.stratumxtime.fixed=list(formula=~stratum*time,fixed=list(time=4,value=1))
model.list=rbind(model.list,c(S="S.stratumxtime",p="p.stratumxtime.fixed",
             Psi="Psi.sxtime"))
model.list=rbind(model.list,c(S="S.stratum",p="p.stratumxtime",Psi="Psi.s"))
#
# Run the list of models
#
mstrata.results=mark.wrapper(model.list,data=mstrata.processed,ddl=mstrata.ddl)
```

```
#
# Export data and models for import to MARK
#
export.MARK(mstrata.processed,"mstrata",mstrata.results,replace=TRUE)
#
# Return model table and list of models
#
return(mstrata.results)
}
mstrata.results=run.mstrata()

data(mallard)
Dot=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~1)))
mallard.proc=process.data(mallard,nocc=90,model="Nest")
export.MARK(mallard.proc,"mallard",Dot,replace=TRUE)
data(robust)
time.intervals=c(0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0)
S.time=list(formula=~time)
p.time.session=list(formula=~-1+session:time,share=TRUE)
GammaDoublePrime.random=list(formula=~time,share=TRUE)
model.1=mark(data = robust, model = "Robust",
time.intervals=time.intervals,
model.parameters=list(S=S.time,
GammaDoublePrime=GammaDoublePrime.random,p=p.time.session))
robust.proc=process.data(data = robust, model = "Robust",
time.intervals=time.intervals)
export.MARK(robust.proc,"robust",model.1,replace=TRUE)
```

`export.model` *Export output files for appending into MARK .dbf/.fpt format*

Description

Creates renamed versions of the output,vcv and residual files so they can be appended into a MARK .dbf file.

Usage

```
export.model(model, replace = FALSE)
```

Arguments

`model`	a mark model object or marklist object
`replace`	if file exists and replace=TRUE, file will be over-written

Details

If `model` is a marklist then it exports each model in the marklist. The function simply copies the files with new names so the MARK interface will recognize them. The marknnn.out is copied as marknnnY.tmp, marknnn.res is copied as marknnnx.tmp and marknnn.vcv is copied as marknnnV.tmp. You can create a MARK .dbf by using `export.chdata` to create an input file for MARK, opening MARK (MARKINT.EXE) to create a new .dbf with the input file, and then using the Output/Append to select the output file (marknnnY.tmp) to append the model with its files. Then you can use any facilities of MARK that are not already included in RMark.

Warning Make sure that you use the .inp created by `export.chdata` with your processed data to create the MARK .dbf file rather than using a separate similar .inp file. It is essential that the group structure and ordering of groups matches between the .inp file and the exported models or you can get erroneous results.

Value

None

Author(s)

Jeff Laake

See Also

`export.chdata`

Examples

```
data(dipper)
mymodel=mark(dipper)
export.model(mymodel,replace=TRUE)
```

---

extract.indices *Various utility functions*

Description

Miscellaneous set of functions that can be used with results from the package.

Usage

```
extract.indices(model,parameter,df)

        nat.surv(model,df)

         pop.est(ns,ps,design,p.vcv)

         compute.Sn(x,df,criterion)
```

```
logitCI(x,se)

search.output.files(x,string)
```

Arguments

model	a mark model object
parameter	character string for a type of parameter for that model (eg, "Phi","p")
df	dataframe containing the columns group, row, column which specify the group number, the row number and column number of the PIM
ns	vector of counts of animals captured
ps	vector of capture probability estimates which match counts
design	design matrix that specifies how counts will be aggregate
p.vcv	variance-covariance matrix for capture probability estimates
x	marklist of models for compute.Sn and a vector of real estimates for logitCI
se	vector of std errors for real estimates
criterion	vector of model selection criterion values (eg AICc)
string	string to be found in output files contained in models in x

Details

Function extract.indices extracts the parameter indices from the parameter index matrices (PIMS) for a particular type of parameter that match a set of group numbers and rows and columns that are defined in the dataframe df. It returns a vector of indices which can be used to specify the set of real parameters to be extracted by covariate.predictions using the index column in data or the indices argument. If df is NULL, it returns a dataframe with all of the indices with model.index being the unique index across all parameters and the par.index which is an index to the row in the design data. If parameter is NULL then the the dataframe is given for all of the parameters.

Function nat.surv produces estimates of natural survival (Sn) from total survival (S) and recovery rate (r) from a joint live-dead model in which all harvest recoveries are reported. In that case, Taylor et al 2005 suggest the following estimator of natural survival Sn=S + (1-S)*r. The arguments for the function are a mark model object and a dataframe df that defines the set of groups and times (row,col) for the natural survival computations. It returns a list with elements: 1) Sn - a vector of estimates for natural survival; one for each entry in df and 2) vcv - a variance-covariance matrix for the estimates of natural survival.

Function pop.est produces estimates of abundance using a vector of counts of animals captured (ns) and estimates of capture probabilities (ps). The estimates can be aggregated or averaged using the design matrix argument. If individual estimates are needed, use an nxn identity matrix for design where n is the length of ns. To get a total of all the estimates use a nx1 column matrix of 1s. Any other design matrix can be specified to subset, aggregate and/or average the estimates. The argument p.vcv is needed to compute the variance-covariance matrix for the abundance estimates using the formula described in Taylor et al. (2002). The function returns a list with elements: 1) Nhat - a vector of abundance estimates and 2) vcv - variance-covariance matrix for the abundance estimates.

Function Compute.Sn creates list structure for natural survival using nat.surv to be used for model averaging natural survival estimates (e.g., model.average(compute.Sn(x,df,criterion))). It returns a list with elements estimates, vcv, weight: 1) estimates - matrix of estimates of natural survival, 2)vcv - list of var-cov matrix for the estimates, and 3) weight - vector of model weights.

Function search.output.filessearches for occurrence of a specific string in output files associated with models in a marklist x. It returns a vector of model numbers in the marklist which have an output file containing the string.

Author(s)

Jeff Laake

References

TAYLOR, M. K., J. LAAKE, H. D. CLUFF, M. RAMSAY and F. MESSIER. 2002. Managing the risk from hunting for the Viscount Melville Sound polar bear population. Ursus 13: 185-202.

TAYLOR, M. K., J. LAAKE, P. D. MCLOUGHLIN, E. W. BORN, H. D. CLUFF, S. H. FERGUSON, A. ROSING-ASVID, R. SCHWEINSBURG and F. MESSIER. 2005. Demography and viability of a hunted population of polar bears. Arctic 58: 203-214.

Examples

```
# Example of computing N-hat for occasions 2 to 7 for the p=~time model
data(dipper)
md=mark(dipper,model.parameters=list(p=list(formula=~time),
       Phi=list(formula=~1)))
# Create a matrix from the capture history strings
xmat=matrix(as.numeric(unlist(strsplit(dipper$ch,""))),
      ncol=nchar(dipper$ch[1]))
# sum number of captures in each column but don't use the first
# column because p[1] can't be estimated
ns=colSums(xmat)[-1]
# extract the indices and then get covariate predictions for p(2),...,p(7)
# which are row-colums 1-6 in PIM for p
p.indices=extract.indices(md,"p",df=data.frame(group=rep(1,6),
   row=1:6,col=1:6))
p.list=covariate.predictions(md,data=data.frame(index=p.indices))
# call pop.est using diagonal design matrix to get
# separate estimate for each occasion
pop.est(ns,p.list$estimates$estimate,
  design=diag(1,ncol=6,nrow=6),p.list$vcv)
```

---

extract.mark.output *Extract results from MARK output file (internal use)*

Description

Extracts the lnl, AICc, npar, beta and real estimates and returns a list of these results for inclusion in the mark object. The elements beta and real are dataframes with fields estimate,se,lcl,ucl. This function was written for internal use and is called by run.mark.model. It is documented here for more advanced users that might want to modify the code or adapt for their own use.

Usage

```
extract.mark.output(out, model, adjust, realvcv = FALSE, vcvfile)
```

Arguments

out	output from MARK analysis (model$output)
model	mark model object
adjust	if TRUE, adjusts number of parameters (npar) to number of columns in design matrix, modifies AIC and records both
realvcv	if TRUE the vcv matrix of the real parameters is extracted and stored in the model results
vcvfile	name of vcv file output

Value

result: list of extracted output elements

lnl	-2xLog-likelihood
deviance	Difference between saturated model and lnl
npar	Number of model parameters
AICc	Small-sample corrected AIC value using npar and n
npar.unadjusted	Number of model parameters as reported by MARK if npar was adjusted
AICc.unadjusted	Small-sample corrected AIC value using npar.unadjusted and n
n	Effective sample size reported by MARK; used in AICc calculation
beta	Dataframe of beta parameters with fields: estimate, se, lcl, ucl
real	Dataframe of real parameters with fields: estimate, se, lcl, ucl
derived.vcv	variance-covariance matrix for derived parameters if any
covariate.values	dataframe with fields Variable and Value which are the covariate names and value used for real parameter estimates in the MARK output
singular	indices of beta parameters that are non-estimable or at a boundary
real.vcv	variance-covariance matrix for real parameters (simplified) if realvcv=TRUE

Author(s)

Jeff Laake

See Also

run.mark.model

`fill.covariates` *Fill covariate entries in MARK design matrix with values*

Description

Replaces covariate names in design matrix with specific values to compute estimates of real parameters at those values using the dataframe from `find.covariates` after any value replacement.

Usage

```
fill.covariates(model, values)
```

Arguments

`model`	MARK model object
`values`	a dataframe matching structure of output from find.covariates with the user-defined values entered

Details

The design matrix for a MARK model with individual covariates contains the covariate names used in the model. In computing the real parameters for the encounter history of an individual it replaces instances of covariate names with the individual covariate values. This function replaces the cells in the design matrix that contain individidual covariates with user-specified values which is an edited version (if needed) of the dataframe returned by `find.covariates`.

Value

New design matrix with user-defined covariate values entered in place of covariate names

Author(s)

Jeff Laake

See Also

`find.covariates`, `compute.real`

Examples

```
data(dipper)
dipper$nsex=as.numeric(dipper$sex)-1
dipper$weight=rnorm(294)
#NOTE:  This generates random valules for the weights so the answers using
# ~weight will vary each time it is run
mod=mark(dipper,model.parameters=list(Phi=list(formula=~nsex+weight)))
# Show approach using individual calls to find.covariates, fill.covariates
# and compute.real
fc=find.covariates(mod,dipper)
fc$value[fc$var=="nsex"]=0 # assign sex value to Female
design=fill.covariates(mod,fc) # fill design matrix with values
# compute and output survivals for females at average weight
female.survival=compute.real(mod,design=design)[1,]
female.survival
# Next show same thing with a call to compute.real and a data frame for
# females and then males
# compute and output survivals for females at average weight
female.survival=compute.real(mod,data=
      data.frame(nsex=0,weight=mean(dipper$weight)))[1,]
female.survival
male.survival=compute.real(mod,data=data.frame(nsex=1,
         weight=mean(dipper$weight)))[1,]
male.survival
# Fit model using sex as a group/factor variable and
# compute v-c matrix for estimates
mod=mark(dipper,groups="sex",
     model.parameters=list(Phi=list(formula=~sex+weight)))
survival.by.sex=compute.real(mod,data=dipper,vcv=TRUE)
survival.by.sex$real[1:2]  # estimates
survival.by.sex$se.real[1:2] # std errors
survival.by.sex$vcv.real[1:2,1:2] # v-c matrix
survival.by.sex$vcv.real[1,2]/prod(survival.by.sex$se.real[1:2])
# sampling correlation of the estimates
```

`find.covariates` *Find covariates in MARK design matrix*

Description

Finds and extracts cells in MARK design matrix containing covariates. Computes mean values of the covariates and assigns those as default values. Returns dataframe that can be edited to replace default values which are then inserted into the design matrix with `fill.covariates` to enable computation of estimates of real parameters with `compute.real`.

Usage

```
find.covariates(model, data = NULL, usemean = TRUE)
```

Arguments

model	MARK model object
data	dataframe used to construct MARK model object; not processed data list
usemean	logical; if TRUE uses mean value of covariate for default and otherwise uses 0

Details

The design matrix for a MARK model with individual covariates contains entries with the covariate names used in the model. In computing the real parameters for the encounter history of an individual it replaces instances of covariate names with the individual covariate values. This function finds all of the cells in the design matrix that contain individidual covariates and constructs a dataframe of the name of the real parameter, the position (row, col) in the design matrix and a default value for the covariate. The default field value is assigned to one of three values in the following priority order: 1) the mean value for the covariates in data (if data is not NULL), 2) the mean values used in the MARK output (if data=NULL,usemean=TRUE), 3) 0 (if usemean=FALSE and data=NULL). The values can also be modified using `fc=edit(fc)` where `fc` is the value from this function.

Value

A dataframe with the following fields

rnames	name of real parameter
row	row number in design matrix (equivalent to `parm.indices` in call to `compute.real`
col	column number in design matrix
var	name of covariate
value	value for covariate

Author(s)

Jeff Laake

See Also

`fill.covariates`, `compute.real`

Examples

```
# see examples in fill.covariates
```

get.link *Compute sets of link values for real parameters*

Description

Computes link values for real parameters for a particular type of parameter (parameter) and returns in a table (dataframe) format.

Usage

```
get.link(model, parameter, beta = NULL, design = NULL, data = NULL,
  vcv = FALSE)
```

Arguments

model	MARK model object
parameter	type of parameter in model (character) (e.g.,"Phi")
beta	values of beta parameters for computation of link values
design	a numeric design matrix with any covariate values filled in with numerical values
data	covariate data to be averaged for estimates if design=NULL
vcv	if TRUE computes and returns the v-c matrix of the subset of the link values

Details

This function is very similar to get.real except that it provides estimates of link values before they are transformed to real estimates using the inverse-link. Also, the value is always a dataframe for the estimates and design data and optionally a variance-covariance matrix. See get.real for further details about the arguments.

Value

estimates: If vcv=TRUE, a list is returned with elements vcv.link and the dataframe estimates. If vcv=FALSE, only the estimates dataframe is returned which has the same structure as in get.real.

Author(s)

Jeff Laake

See Also

compute.link,get.real

get.real *Extract or compute sets of real parameters*

Description

Extracts or computes real parameters for a particular type of parameter (parameter) and returns in either a table (dataframe) format or in PIM format.

Usage

```
get.real(model, parameter, beta = NULL, se = FALSE, design = NULL,
  data = NULL, vcv = FALSE, show.fixed = TRUE, expand = FALSE)
```

Arguments

model	MARK model object
parameter	type of parameter in model (character) (e.g.,"Phi")
beta	values of beta parameters for computation of real parameters
se	if TRUE uses table format and extracts se and confidence intervals, if FALSE uses PIM format with estimates only
design	a numeric design matrix with any covariate values filled in with numerical values
data	covariate data to be averaged for estimates if design=NULL
vcv	if TRUE computes and returns the v-c matrix of the subset of the real parameters
show.fixed	if TRUE fixed values are returned rather than NA in place of fixed values
expand	if TRUE, returns vcv matrix for unique parameters and only simplified unique parameters if FALSE

Details

This function is called by summary.mark to extract(from model$results$real) sets of parameters for display. But, it can also be useful to compute particular sets of real parameters for output, manipulation or plotting etc. It is closely related to compute.real and it uses that function when it computes (rather than extracts) real parameters. It provides an easier way to extract/compute real estimates of a particular type (parameter).

The real parameter estimates are computed when either 1) model$chat > 1, 2) design, data, or beta are are specified with non-NULL values for those arguments, or 3) vcv=TRUE. If none of the above hold, then the estimates are extracted.

If se=FALSE and estimates are shown in triangular or square PIM structure depending on the model and parameter. For triangular, the lower half of the matrix is shown as NA (not applicable in this case). If se=TRUE, the estimate, standard error and confidence interval for the real parameters with the accompanying design data are combined into a dataframe.

If the model contains individual covariates, there are 3 options for specifying the covariate values that are used for the real estimates. If neither design nor data are specified, then the real estimates are computed with the average covariate values used in the MARK output. This is what

is done in the call from summary.mark. Alternatively, the argument design can be given a numeric design matrix for the model with the covariates given specific values. This can be done with find.covariates and fill.covariates. Finally, a quicker approach is to specify a dataframe for data which is averaged for the numeric covariate values and these are automatically filled into the design matrix of the model with calls to find.covariates and fill.covariates from compute.real which is used for computation. The second and third options are essentially the same but creating the completed design matrix will be quicker if multiple calls are made with the same completed design matrix for different parameters. The dataframe for data can contain a single entry to specify certain values for computation.

Value

estimates: if se=FALSE and Beta=NULL, a matrix of estimates or list of matrices for more than one group, and if se=TRUE or beta=is not NULL and vcv=FALSE a dataframe of estimates with attached design data. If vcv=TRUE, a list is returned with elements vcv.real and the dataframe estimates as returned with se=TRUE.

Author(s)

Jeff Laake

See Also

summary.mark,compute.real

Examples

```
data(example.data)
pregion=list(formula=~region)
PhiAge=list(formula=~Age)
mod=mark(example.data,model.parameters=list(p=pregion,Phi=PhiAge),
 groups=c("sex","age","region"),age.var=2,initial.ages=c(0,1,2))
# extract list of Phi parameter estimates for all groups in PIM format
Phi.estimates=get.real(mod,"Phi")
# print out parameter estimates in triangular PIM format
for(i in 1:length(Phi.estimates))
{
  cat(names(Phi.estimates)[i],"\n")
  print(Phi.estimates[[i]]$pim,na.print="")
}
require(plotrix)
#extract parameter estimates of capture probability p with se and conf intervals
p.table=get.real(mod,"p",se=TRUE)
print(p.table[p.table$region==1,])  # print values from region 1
estimates=by(p.table$estimate,p.table$region,mean)
lcl=by(p.table$lcl,p.table$region,mean)
ucl=by(p.table$ucl,p.table$region,mean)
plotCI(c(1:4),estimates,ucl-estimates,estimates-lcl,xlab="Region",
         ylab="Capture probability",
ylim=c(.5,1),main="Capture probability estimates by region")
```

IELogitNormalMR — *Example of Immigration-Emigration LogitNormal Mark-Resight model*

Description

Data and example illustrating Immigration-Emigration LogitNormal Mark-Resight model

Format

A data frame with 34 observations on the following variable.

ch a character vector

Examples

```
data(IELogitNormalMR)
IElogitNor.proc=process.data(IELogitNormalMR,model="IELogitNormalMR",
counts=list("Marked Superpopulation"=c(28, 29, 30, 30, 30, 33, 33, 33, 33, 34, 34, 34),
"Unmarked Seen"=c(264, 161, 152, 217, 217, 160, 195, 159, 166, 152, 175, 190),
"Marked Unidentified"=c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)),
         time.intervals=c(0,0,0,1,0,0,0,1,0,0,0))
IElogitNor.ddl=make.design.data(IElogitNor.proc)
mod1=mark(IElogitNor.proc,IElogitNor.ddl,
model.parameters=list(p=list(formula=~-1+session),
              sigma=list(formula=~session),
  alpha=list(formula=~-1+session:time),
  Nstar=list(formula=~session),
  Nbar=list(formula=~session)))
summary(mod1)

mod2=mark(IElogitNor.proc,IElogitNor.ddl,
  model.parameters=list(p=list(formula=~-1+session:time),
  sigma=list(formula=~session),
  alpha=list(formula=~-1+session:time),
  Nstar=list(formula=~session),
  Nbar=list(formula=~session)),
  initial=mod1)
summary(mod2)
```

import.chdata — *Import capture-recapture data sets from space or tab-delimited files*

Description

A relatively flexible function to import capture history data sets that include a capture (encounter) history read in as a character string and an arbitrary number of user specified covariates for the analysis.

Usage

```
import.chdata(filename, header = TRUE, field.names = NULL,
  field.types = NULL, use.comments = TRUE)
```

Arguments

filename	file name and path for file to be imported; fields in file should be space or tab-delimited
header	TRUE/FALSE; if TRUE first line is name of variables
field.names	vector of field names if header=FALSE; first field should always be ch - capture history remaining number of fields and their names are arbitrary
field.types	vector identifying whether fields (beyond ch) are numeric ("n") or factor ("f") or should be skipped ("s")
use.comments	if TRUE values within /* and */ on data lines are used as row.names for the RMark dataframe. Only use this option if they are unique values.

Details

This function was written both to be a useful tool to import data and as an example for more specific import functions that a user may want to write for data files that do not satisfy the requirements of this function. In particular this function will not handle files with fixed-width format files that do not contain appropriate tab or space delimiters between the fields. It also requires that the first field is the capture (encounter) history which is named "ch" and is a character string. The remaining fields are arbitrary in number and type and are user defined based on the arguments to the functions. Variables that will be used for grouping should be defined with the `field.type="f"`. Numeric individual covariates (e.g., weight) should be input as `field.type="n"`. Fields in the file that should not be imported should be assigned `field.type="s"`. The examples below illustrate different uses of the calling arguments to import several different data sets that meet the modest requirements of this function.

If you specify a frequency for the encounter history, the field name must be `freq`. If you use any other name or spelling it will not be recognized and the default frequency of 1 will be used for each encounter history. This function should not be used with files structured for input into the MARK interface. To use those types of files, see `convert.inp`. It is not neccessary to use either function to create a dataframe for RMark. All you need to is create a dataframe that meets the specification of the RMark format. For example, if you are simulating data, you only need to create a dataframe with the fields ch, freq (if differs from 1) and any covariates you want and then you can use `process.data` on the dataframe.

If you have comments in your data file, they should not have a column header (field name in first row). If `use.comments=TRUE` the comments are used as row names of the data frame and they must be unique. If `use.comments=FALSE` and the file contains comments they are stripped out.

Value

A dataframe for use in MARK analysis with obligate ch character field representing the capture (encounter) history and optional covariate/grouping variables.

Author(s)

Jeff Laake

See Also

export.chdata

Examples

```
pathtodata=paste(path.package("RMark"),"extdata",sep="/")
example.data<-import.chdata(paste(pathtodata,"example.data.txt",sep="/"),
      field.types=c("n","f","f","f"))
edwards.eberhardt<-import.chdata(paste(pathtodata,"EdwardsandEberhardt.txt",
      sep="/"),field.names="ch",header=FALSE)
dipper<-import.chdata(paste(pathtodata,"dipper.txt",sep="/"),
      field.names=c("ch","sex"),header=FALSE)
```

---

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

---

killdeer *Killdeer nest survival example data*

Description

A data set on killdeer that accompanies MARK as an example analysis for the nest survival model.

Format

A data frame with 18 observations on the following 6 variables.

id a MARK comment field with a nest id

FirstFound the day the nest was first found

LastPresent the last day that chicks were present

LastChecked the last day the nest was checked

Fate the fate of the nest; 0=hatch and 1 depredated

Freq the frequency of nests with this data; usually 1

Details

This is a data set that accompanies program MARK as an example for nest survival. The data structure for the nest survival model is completely different from the capture history structure used for most MARK models. To cope with these data you must import them into a dataframe using R commands and assign the specific variable names shown above. The id and Freq fields are optional. Freq is assumed to be 1 if not given. You cannot import the MARK .inp file structure directly into R without some manipulation. Also note that `import.chdata` and `convert.inp` do NOT work for nest survival data. In the examples section below, the first section of code provides an example of converting the killdeer.inp file into a dataframe for RMark.

If your dataframe contains a variable AgeDay1, which is the age of the nest on the first occasion then you can use a variable called NestAge which will create a set of time-dependent covariates named NestAge1,NestAge2 ...NestAge(nocc-1) which will provide a way to incorporate the age of the nest in the model. This was added because the age covariate in the design data for S assumes all nests are the same age and is not particularly useful. This effect could be incorporated by using the add() function in the design matrix but RMark does not have any capability for doing that and it is easier to create a time-dependent covariate to do the same thing.

Examples

```
# EXAMPLE CODE FOR CONVERSION OF .INP TO NECESSARY DATA STRUCTURE
# read in killdeer.inp file
#killdeer=scan("killdeer.inp",what="character",sep="\n")
# strip out ; and write out all but first 2 lines which contain comments
#write(sub(";","",killdeer[3:20]),"killdeer.txt")
# read in as a dataframe and assign names
#killdeer=read.table("killdeer.txt")
#names(killdeer)=c("id","FirstFound","LastPresent","LastChecked","Fate","Freq")
#
# EXAMPLE CODE TO RUN MODELS CONTAINED IN THE MARK KILLDEER.DBF
data(killdeer)
# produce summary
summary(killdeer)
# Define function to run models that are in killdeer.dbf
# You must specify either the number of occasions (nocc) or the time.intervals
# between the occasions.
run.killdeer=function()
{
   Sdot=mark(killdeer,model="Nest",nocc=40)
   STime=mark(killdeer,model="Nest",
       model.parameters=list(S=list(formula=~I(Time+1))),nocc=40,threads=2)
   STimesq=mark(killdeer,model="Nest",
      model.parameters=list(S=list(formula=~I(Time+1)+I((Time+1)^2))),nocc=40,threads=2)
   STime3=mark(killdeer,model="Nest",
      model.parameters=list(S=list(formula=~I(Time+1)+I((Time+1)^2)+I((Time+1)^3))),
                  nocc=40,threads=2)
   return(collect.models())
}
# run defined models
killdeer.results=run.killdeer()
```

larksparrow *Lark Sparrow*

Description

An example of Multiple Scale Occupancy model for some lark sparrow data that was contributed by Davi Pavalacky at Rocky Mountain bird observatory. The study design was a GRTS selection of paired "Deferred" and "Grazed" pastures. The point count locations within each pasture were a random selection of systematic point count locations separated by 125 m. A removal design was used to the set the data to missing after the first detection. The point count data were set to missing when fewer than nine points were surveyed.

Format

A data frame with 52 observations on the following 20 variables.

ch a character vector containing the encounter history

ceap a factor with two levels "Deferred" and "Grazed" corresponding to a rest rotation grazing system with pastures either rested (Deferred) or grazed (Grazed) during the spring bird breeding season.

cwx a continuous covariate for primary occasion x, representing an ocular estimate of the proportion of area covered by crested wheatgrass in a 50-m radius around the point count location.

tdx a continuous covariate for primary occasion x, representing the starting time (h) of each 6-min point count survey measured on the ratio scale (1.5 h = 1 h 30 min).

Examples

```
# Create dataframe
data(LASP)
mscale=LASP

# Process data with MultScalOcc model and use group variables

mscale.proc=process.data(mscale,model="MultScalOcc",groups=c("ceap"),begin.time=1,mixtures=3)

# Create design data

ddl=make.design.data(mscale.proc)

# Create function to build models

do.Species=function()
{
p.1=list(formula=~1)
```

```
p.2=list(formula=~ceap)
p.3=list(formula=~td)

Theta.1=list(formula=~1)
Theta.2=list(formula=~ceap)
Theta.3=list(formula=~cw)

Psi.1=list(formula=~1)
Psi.2=list(formula=~ceap)

cml=create.model.list("MultScalOcc")
return(mark.wrapper(cml,data=mscale.proc,ddl=ddl,adjust=F,realvcv=T))
}

# Run function to get results

Species.results=do.Species()

# Output model table and estimates

Species.results$model.table

Species.results[[as.numeric(rownames(Species.results$model.table[1,]))]]$results$real
Species.results[[as.numeric(rownames(Species.results$model.table[1,]))]]$results$beta

write.csv(Species.results$model.table,file="lasp_model_selection.csv",row.names=F)

write.csv(Species.results[[as.numeric(rownames(Species.results$model.table[1,]))]]$results$real,
 file="lasp_m1_real.csv")
write.csv(Species.results[[as.numeric(rownames(Species.results$model.table[1,]))]]$results$beta,
 file="lasp_m1_beta.csv")

# Covariate prediction and model averaging

# p(time of day)

mintd <- min(mscale[,12:20])
maxtd <- max(mscale[,12:20])
td.values <- mintd+(0:100)*(maxtd-mintd)/100

PIMS(Species.results[[1]],"p",simplified=F)

td <- covariate.predictions(Species.results,data=data.frame(td1=td.values),indices=c(21))

write.table(td$estimates,file="lasp_cov_pred_p_td.csv",sep=",",col.names=T,row.names=F)

# Theta(crested wheatgrass cover)

mincw <- min(mscale[,3:11])
maxcw <- max(mscale[,3:11])
cw.values <- mincw+(0:100)*(maxcw-mincw)/100

PIMS(Species.results[[1]],"Theta",simplified=F)
```

```
cw <- covariate.predictions(Species.results,data=data.frame(cw1=cw.values),indices=c(3))

write.table(cw$estimates,file="lasp_cov_pred_theta_cw.csv",sep=",",col.names=T,row.names=F)

# Psi(ceap grazing for wildlife practice)

ceap.values <- as.data.frame(matrix(c(1,2),ncol=1))
names(ceap.values) <- c("index")

PIMS(Species.results[[1]],"Psi",simplified=F)

ceap <- covariate.predictions(Species.results,data=data.frame(ceap=ceap.values))

write.table(ceap$estimates,file="lasp_cov_pred_psi_ceap.csv",sep=",",col.names=T,row.names=F)
```

`load.model` *Load external model*

Description

Loads external model into workspace with name model

Usage

```
load.model(model)
```

Arguments

`model`	name of MARK model object stored externally

Value

None; side effect only to load object with name model (not name given to model)

Author(s)

Jeff Laake

LogitNormalMR *Example of LogitNormal Mark-Resight model*

Description

Data and example illustrating LogitNormal Mark-Resight model.

Format

A data frame with 35 observations on the following variable.

ch a character vector

Examples

```
data(LogitNormalMR)
logitNor.proc=process.data(LogitNormalMR,model="LogitNormalMR",
counts=list("Unmarked seen"=c(96,68,59),
    "Marked Unidentified"=c(0,0,0,0,1,1,1,0,0,3,0,1)),
         time.intervals=c(0,0,0,1,0,0,0,1,0,0,0))
logitNor.ddl=make.design.data(logitNor.proc)
mod=mark(logitNor.proc,logitNor.ddl,threads=2)
summary(mod)
```

make.design.data *Create design dataframes for MARK model specification*

Description

For each type of parameter in the analysis model (e.g, p, Phi, r), this function creates design data based on the parameter index matrix (PIM) in MARK terminology. Design data relate parameters to the sampling and data structures; whereas data relate to the object(animal) being sampled. Design data relate parameters to time, age, cohort and group structure. Any variables in the design data can be used in formulas to specify the model in make.mark.model.

Usage

```
make.design.data(data, parameters = list(), remove.unused = FALSE,
  right = TRUE, common.zero = FALSE)
```

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

subtract.stratum	a vector of strata letters (one for each strata) that specifies the tostratum that is computed by subtraction for mlogit parameters like Psi
age.bins	bins for binning ages
time.bins	bins for binning times
cohort.bins	bins for binning cohorts
pim.type	either "all" for all different, "time" for column time structure, or "constant" for all values the same within the PIM

remove.unused	If TRUE, unused design data are deleted; see details below
right	If TRUE, bin intervals are closed on the right
common.zero	if TRUE, uses a common begin.time to set origin (0) for Time variable defaults to FALSE for legacy reasons but should be set to TRUE for models that share formula like p and c with the Time model

Details

After processing the data, the next step is to create the design data for building the models which is done with this function. The design data are different than the capture history data that relates to animals. The types of parameters and design data are specific to the type of analysis. For example, consider a CJS analysis that has parameters Phi and p. If there are 4 occasions, there are 3 cohorts and potentially 6 different Phi and 6 different p parameters for a single group. The format for each parameter information matrix (PIM) in MARK is triangular. RMark uses the all different formulation for PIMS by default, so the PIMs would be

```
 Phi p 1 2 3 7 8 9 4 5 10 11 6 12
```

If you chose pim.type="time" for each parameter in "CJS", then the PIMS are structured as

```
 Phi p 1 2 3 4 5 6 2 3 5 6 3 6
```

That structure is only useful if there is only a single release cohort represented by the PIM. If you choose this option and there is more than one cohort represented by the PIM then it will restrict the possible set of models that can be represented.

Each of these parameters relates to different times, different cohorts (time of initial release) and different ages (at least in terms of time since first capture). Thus we can think of a data frame for each parameter that might look as follows for Phi for the all different structure:

```
 Index time cohort age 1 1 1 0 2 2 1 1 3 3 1 2 4 2 2 0 5 3 2 1
6 3 3 0
```

With this design data, one can envision models that describe Phi in terms of the variables time, cohort and age. For example a time model would have a design matrix like:

```
 Int T2 T3 1 1 0 0 2 1 1 0 3
1 0 1 4 1 1 0 5 1 0 1 6 1 0 1
```

Or a time + cohort model might look like

```
 Int T2 T3 C2 C3 1 1 0 0 0 0 2 1 1 0 0 0 3 1 0 1 0 0 4 1 1 0 1
0 5 1 0 1 1 0 6 1 0 1 0 1
```

While you could certainly develop these designs manually within MARK, the power of the R code rests with the internal R function `model.matrix` which can take the design data and create the design matrix from a formula specification such as `~time` or `~time+cohort` alleviating the need to create the design matrix manually. While many analyses may only need age, time or cohort, it is quite possible to extend the kind of design data, to include different functions of these variables or add additional variables such as effort. One could consider design data for p as follows:

```
 Index time
cohort age effort juvenile 7 1 1 1 10 1 8 2 1 2 5 0 9 3 1 3 15 0 10 2 2 1 5
1 11 3 2 2 15 0 12 3 3 1 15 1
```

The added columns represent a time dependent covariate (effort) and an age variable of juvenile/adult. With these design data, it is easy to specify different models such as `~time`, `~effort`, `~effort+age` or `~effort+juvenile`.

With the simplest call:

```
ddl=make.design.data(proc.example.data)
```

the function creates default design data for each type of parameter in the model as defined by `proc.example.data$model`. If `proc.example.data` was created with the call in the first example of `process.data`, the model is "CJS" (the default model) so the return value is a list with 2 data frames: one for Phi and one for p. They can be accessed as `ddl$Phi` (or `ddl[["Phi"]]`) and `ddl$p` (or `ddl[["p"]]`) or as `ddl[[1]]` and `ddl[[2]]` respectively. Using the former notation is easier and safer because it is independent of the ordering of the parameters in the list. For this example, there are 16 groups and each group has 21 Phi parameters and 21 p parameters. Thus, there are 336 rows (parameters) in the design data frame for both Phi and p and thus a total of 772 parameters.

The default fields in each dataframe are `group`, `cohort`, `age`, `time`, `Cohort`, `Age`, and `Time`. The first 4 fields are factor variables, whereas `Cohort`, `Age` and `Time` are numeric covariate versions of `cohort`, `age`, and `time` shifted so the first value is always zero. In addition, for closed capture heterogeneity models a factor variable `mixture` is included. If `groups` were created in the call to `process.data`, then the factor variables used to create the `groups` are also included in the design data for each type of parameter. If one of the grouping variables is an age variable it is named `initial.age.class` to recognize explicitly that it represents a static initial age and to avoid naming conflicts with `age` and `Age` variables which represent dynamic age variables of the age of the animal through time. Non-age related grouping variables are added to the design data using their names in `data`. For example if `proc.example.data` is defined using the first example in `process.data`, then the fields `sex`, `initial.age.class` (in place of `age` in this case), and `region` are added to the design data in addition to the `group` variable that has 16 levels. The levels of the `group` variable are created by pasting (concatenating) the values of the grouping factor in order. For example, M11 is sex=M, age class=1 and region=1.

By default, the factor variables `age`, `time`, and `cohort` are created such that there is a factor level for each unique value. By specfying values for the argument `parameters`, the values of `age`, `time`, and `cohort` can be binned (put into intervals) to reduce the number of levels of each factor variable. The argument `parameters` is specified as a list of lists. The first level of the list specifies the parameter

type and the second level specifies the variables (age, time, or cohort) that will be binned and the cutpoints (endpoints) for the intervals. For example, if you expected that survival may change substantially to age 3 (i.e. first 3 years of life) but remain relatively constant beyond then, you could bin the ages for survival as 0,1,2,3-8. Likewise, as well, you could decide to bin time into 2 intervals for capture probability in which effort and expected capture probability might be constant within each interval. This could be done with the following call using the argument parameters:

```
ddl=make.design.data(proc.example.data,
parameters=list(Phi=list(age.bins=c(0,0.5,1,2,8)),
p=list(time.bins=c(1980,1983,1986))))
```

In the above example, age is binned for Phi but not for p; likewise time is binned for p but not for Phi. The bins for age were defined as 0,0.5,1,2,8 because the intervals are closed ("]" - inclusive) on the right by default and open ("(" non-inclusive) on the left, except for the first interval which is closed on the left. Had we used 0,1,2,8, 0 and 1 would have been in a single interval. Any value less than 1 and greater than 0 could be used in place of 0.5. Alternatively, the same bins could be specified as:

```
ddl=make.design.data(proc.example.data,
parameters=list(Phi=list(age.bins=c(0,1,2,3,8)),
p=list(time.bins=c(1980,1984,1986))),right=FALSE)
```

To create the design data and maintain flexibility, I recommend creating the default design data and then adding other binned variables with the function add.design.data. The 2 binned variables defined above can be added to the default design data as follows:

```
 ddl=make.design.data(proc.example.data)
ddl=add.design.data(proc.example.data,ddl,parameter="Phi",type="age",
bins=c(0,.5,1,2,8),name="agebin")
ddl=add.design.data(proc.example.data,ddl,parameter="p",type="time",
bins=c(1980,1983,1986),name="timebin")
```

Adding the other binned variables to the default design data, allows models based on either time, timebin, or Time for p and age, agebin or Age for Phi. Any number of additional binned variables can be defined as long as they are given unique names. Note that R is case-specific so ~Time specifies a model which is a linear trend over time ((e.g. Phi(T) or p(T) in MARK) whereas ~time creates a model with a different factor level for each time in the data (e.g. Phi(t) or p(t) in MARK) and ~timebin creates a model with 2 factor levels 1980-1983 and 1984-1986.

Some circumstances may require direct manipulation of the design data to create the needed variable when simple binning is not sufficient or when the design data is a variable other than one related to time, age, cohort or group (e.g., effort index). This can be done with any of the vast array of R commands. For example, consider a situation in which 1983 and 1985 were drought years and you wanted to develop a model in which survival was different in drought and non-drought years. This could be done with the following commands:

```
ddl$Phi$drought=0
ddl$Phi$drought[ddl$phi$time==1983 | ddl$Phi$time==1985]= 1
```

The first command creates a variable named drought in the Phi design data and initializes it with 0. The second command changes the drought variable to 1 for the years 1983 and 1985. The single brackets [] index a data frame, matrix or vector. In this case `ddl$Phi$drought` is a vector and `ddl$Phi$time==1983 | ddl$Phi$time==1985` selects the values in which time is equal (==) to 1983 or ("|") 1985. A simpler example might occur if we want to create a function of one of the continuous variables. If we wanted to define a model for p that was a function of age and age squared, we could add the age squared variable as:

```
ddl$p$Agesq=ddl$p$Age^2
```

Any of the fields in the design data can be used in formulae for the parameters. However, it is important to recognize that additional variables you define and add to the design data are specific to a particular type of parameter (e.g., Phi, p, etc). Thus, in the above example, you could not use Agesq in a model for Phi without also adding it to the Phi design data. As described in `make.mark.model`, there is actually a simpler way to add simple functions of variables to a formula without defining them in the design data.

The above manipulations are sufficient if there is only one or two variables that need to be added to the design data. If there are many covariates that are time(occasion)-specific then it may be easier to setup a dataframe with the covariate data and use `merge_design.covariates`.

The fields that are automatically created in the design data depends on the model. For example, with models such as "POPAN" or any of the "Pradel" models, the PIM structure is called square which really means that it is a single row and all the rows are the same length for each group. Thus, rectangular or row may have been a better descriptor. Regardless, in this case there is no concept of a cohort within the PIM which is equivalent to a row within a triangular PIM for "CJS" type models. Thus, for parameters with "Square" PIMS the cohort (and Cohort) field is not generated. The cohort field is also not created if `pim.type="time"` for "Triangular" PIMS, because that structure has the same structure for each row (cohort) and adding cohort effects would be inappropriate.

For models with "Square" PIMS or `pim.type="time"` for "Triangular" PIMS, it is possible to create a cohort variable by defining the cohort variable as a covariate in the capture history data and using it as a variable for creating groups. As with all grouping variables, it is added to the design data. Now the one caution with "Square" PIMS is that they are all the same length. Imagine representing a triangular PIM with a set of square PIMS with each row being a cohort. The resulting set of PIMS is now rectangular but the lower portion of the matrix is superfluous because the parameters represent times prior to the entry of the cohort, assuming that the use of cohort is to represent a birth cohort. This is only problematic for these kinds of models when the structure accomodates age and the concept of a birth cohort. The solution to the problem is to delete the design data for the superfluous parameters after is is created. For example, let us presume that you used cohort with 3 levels as a grouping variable for a model with "Square" PIMS which has 3 occasions. Then, the PIM structure would look as follows for Phi:

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

If each row represented a cohort that entered at occasions 1,2,3 then parameters 4,7,8 are superfluous or could be thought of as representing cells that are structural zeros in the model because no observations can be made of those cohorts at those times.

After creating the design data, the unneeded rows can be deleted with R commands or you can use the argument `remove.unused=TRUE`. As an example, a code segment might look as follows if `chdata` was defined properly:

```
mydata=process.data(chdata,model="POPAN",groups="cohort")
ddl=make.design.data(mydata) ddl$Phi=ddl$Phi[-c(4,7,8),]
```

If cohort and time were suitably defined an easier solution especially for a larger problem would be

```
ddl$Phi=ddl$Phi[as.numeric(ddl$Phi$time)>=as.numeric(ddl$Phi$cohort),]
```

Which would only keep parameters in which the time is the same or greater than the cohort. Note that time and cohort would be factor variables and < and > do not make sense which is the reason for the `as.numeric` which translates the factor to a numeric ordering of factors (1,2,...) but not the numeric value of the factor level (e.g., 1987,1998). Thus, the above assumes that both time and cohort have the same factor levels. The design data is specific to each parameter, so the unneeded parameters need to be deleted from design data of each parameter.

However, all of this is done automatically by setting the argument `remove.unused=TRUE`. It functions differently depending on the type of PIM structure. For models with "Triangular" PIMS, unused design data are determined based on the lack of a release cohort. For example, if there were no capture history data that started with 0 and had a 1 in the second position ("01.....") that would mean that there were no releases on occasion 2 and row 2 in the PIM would not be needed so it would be removed from the design data. If `remove.unused=TRUE` the design data are removed for any missig cohorts within each group. For models with "Square" PIMS, cohort structure is defined by a grouping variable. If there is a field named "cohort" within the design data, then unused design data are defined to occur when time < cohort. This is particularly useful for age structured models which define birth cohorts. In that case there will be sampling times prior to the birth of the cohort which are not relevant and should be treated as "structural zeros" and not as a zero based on stochastic events.

If the design data are removed, when the model is constructed with `make.mark.model`, the argument `default.fixed` controls what happens with the real parameters defined by the missing design data. If `default.fixed=TRUE`, then the real parameters are fixed at values that explain with certainty the observed data (e.g., p=0). That is necessary for models with "Square" PIMS (eg, POPAN and Pradel models) that include each capture-history position in the probability calculation. For "Triangular" PIMS with "CJS" type models, the capture(encounter) history probability is only computed for occasions past the first "1", the release. Thus, when a cohort is missing there are no entries and the missing design data are truly superfluous and `default.fixed=FALSE` will assign the missing design data to a row in the design matrix which has all 0s. That row will show as a real parameter of (0.5 for a logit link) but it is not included in any parameter count and does not affect any calculation. The advantage in using this approach is that the R code recognizes these and displays blanks for these missing parameters, so it makes for more readable output when say every other cohort is missing. See `make.mark.model` for more information about deleted design data and what this means to development of model formula.

For design data of "Multistrata" models, additional fields are added to represent strata. A separate PIM is created for each stratum for each parameter and this is denoted in the design data with the addition of the factor variable `stratum` which has a level for each stratum. In addition, for each stratum a dummy variable is created with the name of the stratum (`strata.label`)and it has value 1 when the parameter is for that stratum and 0 otherwise. Using these variables with the interaction operator ":" in formula allows more flexibility in creating model structure for some strata and not others. All "Multistrata" models contain "Psi" parameters which represent the transitions from a stratum to all other strata. Thus if there are 3 strata, there are 6 PIMS for the "Psi" parameters to represent transition from A to B, A to C, B to A, B to C, C to A and C to B. The "Psi" parameters are

represented by multimonial logit links and the probability of remaining in the stratum is determined by substraction. To represent these differences, a factor variable `tostratum` is created in addition to `stratum`. Likewise, dummy variables are created for each stratum with names created by pasting "to" and the strata label (e.g., toA, toB etc). Some examples of using these variables to create models for "Psi" are given in `make.mark.model`.

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 as described in the details above.

Author(s)

Jeff Laake

See Also

`process.data`,`merge_design.covariates`, `add.design.data`, `make.mark.model`, `run.mark.model`

Examples

```
data(example.data)
proc.example.data=process.data(example.data)
ddl=make.design.data(proc.example.data)
ddl=add.design.data(proc.example.data,ddl,parameter="Phi",type="age",
  bins=c(0,.5,1,2,8),name="agebin")
ddl=add.design.data(proc.example.data,ddl,parameter="p",type="time",
  bins=c(1980,1983,1986),name="timebin")
```

---

make.mark.model *Create a MARK model for analysis*

Description

Creates a MARK model object that contains a MARK input file with PIMS and design matrix specific to the data and model structure and formulae specified for the model parameters.

Usage

```
make.mark.model(data, ddl, parameters = list(), title = "",
  model.name = NULL, initial = NULL, call = NULL, default.fixed = TRUE,
  options = NULL, profile.int = FALSE, chat = NULL, simplify = TRUE,
  input.links = NULL, parm.specific = FALSE, mlogit0 = FALSE,
  hessian = FALSE, accumulate = TRUE)
```

Arguments

data	Data list resulting from function process.data
ddl	Design data list from function make.design.data
parameters	List of parameter formula specifications
title	Title for the analysis (optional)
model.name	Model name to override default name (optional)
initial	Vector of named or unnamed initial values for beta parameters or previously run model (optional)
call	Pass function call when this function is called from another function (e.g.mark) (internal use)
default.fixed	if TRUE, real parameters for which the design data have been deleted are fixed to default values
options	character string of options for Proc Estimate statement in MARK .inp file
profile.int	if TRUE, requests MARK to compute profile intervals
chat	pre-specified value for chat used by MARK in calculations of model output
simplify	if FALSE, does not simplify PIM structure
input.links	specifies set of link functions for parameters with non-simplified structure
parm.specific	if TRUE, forces a link to be specified for each parameter
mlogit0	if TRUE, any real parameter that is fixed to 0 and has an mlogit link will have its link changed to logit so it can be simplified
hessian	if TRUE specifies to MARK to use hessian rather than second partial matrix
accumulate	if TRUE accumulate like data values into frequencies

Details

This function is called by mark to create the model but it can be called directly to create but not run the model. All of the arguments have default values except for the first 2 that specify the processed data list (data) and the design data list (ddl). If only these 2 arguments are specified default models are used for the parameters. For example, following with the example from process.data and make.design.data, the default model can be created with:

```
mymodel=make.mark.model(proc.example.data,ddl)
```

The call to make.mark.model creates a model object but does not do the analysis. The function returns a list that includes several fields including a design matrix and the MARK input file that will be used with MARK.EXE to run the analysis from function run.mark.model. The following shows the names of the list elements in mymodel:

```
 names(mymodel) [1] "data" "model" "title" "model.name"
"links" [6] "mixtures" "call" "parameters" "input" "number.of.groups" [11]
"group.labels" "nocc" "begin.time" "covariates" "fixed" [16] "design.matrix"
"pims" "design.data" "strata.labels" "mlogit.list" [21] "simplify"
```

The list is defined to be a mark object which is done by assigning a class vector to the list. The classes for an R object can be viewed with the class function as shown below:

```
class(mymodel) [1] "mark" "CJS"
```

Each MARK model has 2 class values. The first identifies it as a mark object and the second identifies the type of mark analysis, which is the default "CJS" (recaptures only) in this case. The use of the class feature has advantages in using generic functions and identifying types of objects. An object of class mark is defined in more detail in function mark.

To fit non-trivial models it is necessary to understand the remaining calling arguments of make.mark.model and R formula notation. The model formulae are specified with the calling argument parameters. It uses a similar list structure as the parameters argument in make.design.data. It expects to get a list with elements named to match the parameters in the particular analysis (e.g., Phi and p in CJS) and each list element is a list, so it is a list of lists). For each parameter, the possible list elements are formula, link, fixed,component, component.name, remove.intercept. In addition, for closed capture models and robust design model, the element share is included in the list for p (capture probabilities) and GammaDoublePrime (respectively) to indicate whether the model is shared (share=TRUE) or not-shared (the default) (share=FALSE) with c (recapture probabilities) and GammaPrime respectively.

formula specifies the model for the parameter using R formula notation. An R formula is denoted with a ~ followed by variables in an equation format possibly using the + , *, and : operators. For example, ~sex+age is an additive model with the main effects of sex and age. Whereas, ~sex*age includes the main effects and the interaction and it is equivalent to the formula specified as ~sex+age+sex:age where sex:age is the interaction term. The model ~age+sex:age is slightly different in that the main effect for sex is dropped which means that intercept of the age effect is common among the sexes but the age pattern could vary between the sexes. The model ~sex*Age which is equivalent to ~sex + Age + sex:Age has only 4 parameters and specifies a linear trend with age and both the intercept and slope vary by sex. One additional operator that can be useful is I() which allows computation of variables on the fly. For example, the addition of the Agesq variable in the design data (as described above) can be avoided by using the notation ~Age + I(Age^2) which specifies use of a linear and quadratic effect for age. Note that specifying the model ~age + I(age^2) would be incorrect and would create an error because age is a factor variable whereas Age is not.

As an example, consider developing a model in which Phi varies by age and p follows a linear trend over time. This model could be specified and run as follows:

```
 p.Time=list(formula=~Time) Phi.age=list(formula=~age)
Model.p.Time.Phi.age=make.mark.model(proc.example.data,ddl,
parameters=list(Phi=Phi.age,p=p.Time))
Model.p.Time.Phi.age=run.mark.model(Model.p.Time.Phi.age)
```

The first 2 commands define the p and Phi models that are used in the parameter list in the call to make.mark.model. This is a good approach for defining models because it clearly documents the models, the definitions can then be used in many calls to make.mark.model and it will allow a variety of models to be developed quickly by creating different combinations of the parameter models. Using the notation above with the period separating the parameter name and the description (eg., p.Time) gives the further advantage that all possible models can be developed quickly with the functions create.model.list and mark.wrapper.

Model formula can use any field in the design data and any individual covariates defined in data. The restrictions on individual covariates that was present in versions before 1.3 have now been

removed. You can now use interactions of individual covariates with all design data covariates and products of individual covariates. You can specify interactions of individual covariates and factor variables in the design data with the formula notation. For example, `~region*x1` describes a model in which the slope of `x1` varies by `region`. Also, `~time*x1` describes a model in which the slope for `x1` varied by time; however, there would be only one value of the covariate per animal so this is not a time varying covariate model. Models with time varying covariates are now more easily described with the improvements in version 1.3 but there are restrictions on how the time varying individual covariates are named. An example would be a trap dependence model in which capture probability on occasion i+1 depends on whether they were captured on occasion i. If there are n occasions in a CJS model, the 0/1 (not caught/caught) for occasions 1 to n-1 would be n-1 individual covariates to describe recapture probability for occasions 2 to n. For times 2 to n, a design data field could be defined such that the variable timex is 1 if time==x and 0 otherwise. The time varying covariates must be named with a time suffix on the base name of the covariate. In this example they would be named as `x2,. . .,xn` and the model could be specified as `~time + x` for time variation and a constant trap effect or as `~time + time:x` for a trap effect that varied by time. If in the `process.data` call, the argument `begin.time` was set to the year 1990, then the variables would have to be named x1991,x1992,... because the first recapture occasion would be 1991. Note that the times are different for different parameters. For example, survival is labeled based on the beginning of the time interval which would be 1990 so the variables should be named appropriately for the parameter model in which they will be used.

In previous versions to handle time varying covariates with a constant effect, it was necessary to use the component feature of the parameter specification to be able to append one or more arbitrary columns to the design matrix. That is no longer required for individual covariates and the component feature was removed in v2.0.8.

There are three other elements of the parameter list that can be useful on occasion. The first is `link` which specifies the link function for transforming between the beta and real parameters. The possible values are "logit", "log", "identity" and "mlogit(*)" where * is a numeric identifier. The "sin" link is not allowed because all models are specified using a design matrix. The typical default values are assigned to each parameter (eg "logit" for probabilities, "log" for N, and "mlogit" for pent in POPAN), so in most cases it will not be necessary to specify a link function.

The second is `fixed` which allows real parameters to be set at fixed values. The values for `fixed` can be either a single value or a list with 5 alternate forms for ease in specifying the fixed parameters. Specifying `fixed=value` will set all parameters of that type to the specified value. For example, `Phi=list(fixed=1)` will set all Phi to 1. This can be useful for situations like specifying F in the Burnham/Barker models to all have the same value of 1. Fixed values can also be specified as a list in which values are specified for certain indices, times, ages, cohorts, and groups. The first 3 will be the most useful. The first list format is the most general and flexible but it requires an understanding of the PIM structure and index numbers for the real parameters. For example,

```
Phi=list(formula=~time, fixed=list(index=c(1,4,7),value=1))
```

specifies Phi varying by time, but the real parameters 1,4,7 are set to 1. The `value` field is either a single constant or its length must match the number of indices. For example,

```
Phi=list(formula=~time, fixed=list(index=c(1,4,7),value=c(1,0,1)))
```

sets real parameters 1 and 7 to 1 and real parameter 4 to 0. Technically, the index/value format for fixed is not wedded to the parameter type (i.e., values for p can be assigned within Phi list), but for the sake of clarity they should be restricted to fixing real parameters associated with the particular parameter type. The `time` and `age` formats for fixed will probably be the most useful. The format

fixed=list(time=x, value=y) will set all real parameters (of that type) for time x to value y. For example,

```
p=list(formula=~time,fixed=list(time=1986,value=1))
```

sets up time varying capture probability but all values of p for 1986 are set to 1. This can be quite useful to set all values to 0 in years with no sampling (e.g.,

```
fixed=list(time=c(1982,1984,1986), value=0)
```

). The age, cohort and group formats work in a similar fashion. It is important to recognize that the value you specify for time, age, cohort and group must match the values in the design data list. This is another reason to add binned fields for age, time etc with add.design.data after creating the default design data with make.design.data. Also note that the values for time and cohort are affected by the begin.time argument specified in process.data. Had I not specified begin.time=1980, to set p in the last occasion (1986), the specification would be

```
p=list(formula=~time,fixed=list(time=7,value=1))
```

because begin.time defaults to 1. The advantage of the time-, age-, and cohort- formats over the index-format is that it will work regardless of the group definition which can easily be changed by changing the groups argument in process.data. The index-format will be dependent on the group structure because the indexing of the PIMS will most likely change with changes in the group structure.

Parameters can also be fixed at default values by deleting the specific rows of the design data. See make.design.data and material below. The default value for fixing parameters for deleted design data can be changed with the default=value in the parameter list.

The final useful element of the parameter list is the remove.intercept argument. It is set to TRUE to forcefully remove the intercept. In R notation this can be done by specifiying the formula notation ~-1+... but in formula with nested interactions of factor variables and additive factor variables the -1 notation will not remove the intercept. It will simply adjust the column definitions but will keep the same number of columns and the model will be overparameterized. The problem occurs with nested factor variables like tostratum within stratum for multistrata designs (see mstrata). As shown in that example, you can build a formula -1+stratum:tostratum to have transitions that are stratum-specific. If however you also want to add a sex effect and you specify -1+sex+stratum:tostratum it will add 2 columns for sex labelled M and F when in fact you only want to add one column because the intercept is already contained within the stratum:tostratum term. The argument remove.intercept will forcefully remove the intercept but it needs to be able to find a column with all 1's. For example, Psi=list(formula=~sex+stratum:tostratum,remove.intercept=TRUE) will work but Psi=list(formula=~-1+sex+stratum:tostratum,remove.intercept=TRUE) will not work. Also, the -1 notation should be used when there is not an added factor variable because

```
Psi=list(formula=~stratum:tostratum,remove.intercept=TRUE)
```

will not work because while the stratum:tostratum effectively includes an intercept it is equivalent to using an identity matrix and is not specified as treatment contrast with one of the columns as all 1's.

The argument simplify determines whether the pims are simplified such that only indices for unique and fixed real parameters are used. For example, with an all different PIM structure with CJS with K occasions there are K*(K-1) real parameters for Phi and p. However, if you use simplify=TRUE with the default model of Phi(.)p(.), the pims are re-indexed to be 1 for all the Phi's and 2 for all

the p's because there are only 2 unique real parameters for that model. Using simplify can speed analysis markedly and probably should always be used. This was left as an argument only to test that the simplification was working and produced the same likelihood and real parameter estimates with and without simplification. It only adjust the rows of the design matrix and not the columns. There are some restrictions for simplification. Real parameters that are given a fixed value are maintained in the design matrix although it does simplify amongst the fixed parameters. For example, if there are 50 real parameters all fixed to a value of 1 and 30 all fixed to a value of 0, they are reduced to 2 real parameters fixed to 1 and 0. Also, real parameters such as Psi in Multistrata and pent in POPAN that use multinomial logits are not simplified because they must maintain the structure created by the multinomial logit link. All other parameters in those models are simplified. The only downside of simplification is that the labels for real parameters in the MARK output are unreliable because there is no longer a single label for the real parameter because it may represent many different real parameters in the all-different PIM structure. This is not a problem with the labels in R because the real parameter estimates are translated back to the all-different PIM structure with the proper labels.

The argument `default.fixed` is related to deletion of design data (see `make.design.data`). If design data are deleted and `default.fixed=T` the missing real parameters are fixed at a reasonable default to represent structural "zeros". For example, p is set to 0, Phi is set to 1, pent is set to 0, etc. For some parameters there are no reasonable values (e.g., N in POPAN), so not all parameters will have defaults. If a parameter does not have a default or if `default.fixed=F` then the row in the design matrix for that parameter is all zeroes and its real value will depend on the link function. For example, with "logit" link the real parameter value will be 0.5 and for the log link it will be 1. As long as the inverse link is defined for 0 it will not matter in those cases in which the real parameter is not used because it represents data that do not exist. For example, in a "CJS" model if initial captures (releases) only occur every other occasion, but recaptures (resightings) occurred every occasion, then every other cohort (row) in the PIM would have no data. Those rows (cohorts) could be deleted from the design data and it would not matter if the real parameter was fixed. However, for the case of a Jolly-Seber type model (eg POPAN or Pradel models) in which the likelihood includes a probability for the leading zeroes and first 1 in a capture history (a likelihood component for the first capture of unmaked animals), and groups represent cohorts that enter through time, you must fix the real parameters for the unused portion of the PIM (ie for occasions prior to time of birth for the cohort) such that the estimated probability of observing the structural 0 is 1. This is easily done by setting the pent (probability of entry) to 0 or by setting the probability of capture to 0 or both. In that case if `default.fixed=F`, the probabilities for all those parameters would be incorrectly set to 0.5 for p and something non-zero but not predetermined for pent because of the multinomial logit. Now it may be possible that the model would correctly estimate these as 0 if the real parameters were kept in the design, but we know what those values are in that case and they need not be estimated. If it is acceptable to set `default.fixed=F`, the functions such as `summary.mark` recognize the non-estimated real parameters and they are not shown in the summaries because in essence they do not exist. If `default.fixed=T` the parameters are displayed with their fixed value and for `summary.mark(mymodel,se=TRUE)`, they are listed as "Fixed".

Missing design data does have implications for specifying formula but only when interactions are specified. With missing design data various factors may not be fully crossed. For example, with 2 factors A and B, each with 2 levels, the data are fully crossed if there are data with values A1&B1, A1&B2, A2&B1 and A2&B2. If data exist for each of the 4 combinations then you can described the interaction model as ~A*B and it will estimate 4 values. However, if data are missing for one of more of the 4 cells then the "interaction" formula should be specified as ~-1+A:B or ~-1+B:A or ~-1+A the combinations with data. An example of this could be a marking program with multiple sites which resighted at all occasions but which only marked at sites on alternating occasions. In that

case time is nested within site and time-site interaction models would be specified as ~-1+time:site.

The argument `title` supplies a character string that is used to label the output. The argument `model.name` is a descriptor for the model used in the analysis. The code constructs a model name from the formula specified in the call (e.g., `Phi(~1)p(~time)`) but on occasion the name may be too long or verbose, so it can be over-ridden with the `model.name` argument.

The final argument `initial` can be used to provide initial estimates for the beta parameters. It is either 1) a single starting value for each parameter, 2) an unnamed vector of values (one for each parameter), 3) named vector of values, or 4) the name of `mark` object that has already been run. For cases 3 and 4, the code only uses appropriate initial beta estimates in which the column names of the design matrix (for model) or vector names match the column names in the design matrix of the model to be run. Any remaining beta parameters without an initial value specified are assigned a 0 initial value. If case 4 is used the models must have the same number of rows in the design matrix and thus presumably the same structure. As long as the vector elements are named (#3), the length of the `initial` vector no longer needs to match the number of parameters in the new model as long as the elements are named. The names can be retrieved either from the column names of the design matrix or from `rownames(x$results$beta)` where `x` is the name of the `mark` object.

`options` is a character string that is tacked onto the `Proc Estimate` statement for the MARK .inp file. It can be used to request options such as NoStandDM (to not standardize the design matrix) or SIMANNEAL (to request use of the simulated annealing optimization method) or any existing or new options that can be set on the estimate proc.

Value

model: a MARK object except for the elements `output` and `results`. See `mark` for a detailed description of the list contents.

Author(s)

Jeff Laake

See Also

`process.data`,`make.design.data`, `run.mark.model` `mark`

Examples

```
data(dipper)
#
# Process data
#
dipper.processed=process.data(dipper,groups=("sex"))
#
# Create default design data
#
dipper.ddl=make.design.data(dipper.processed)
#
# Add Flood covariates for Phi and p that have different values
#
dipper.ddl$Phi$Flood=0
```

```
dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 | dipper.ddl$Phi$time==3]=1
dipper.ddl$p$Flood=0
dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
#
#  Define range of models for Phi
#
Phidot=list(formula=~1)
Phitime=list(formula=~time)
PhiFlood=list(formula=~Flood)
#
#  Define range of models for p
#
pdot=list(formula=~1)
ptime=list(formula=~time)
#
# Make assortment of models
#
dipper.phidot.pdot=make.mark.model(dipper.processed,dipper.ddl,
  parameters=list(Phi=Phidot,p=pdot))
dipper.phidot.ptime=make.mark.model(dipper.processed,dipper.ddl,
  parameters=list(Phi=Phidot,p=ptime))
dipper.phiFlood.pdot=make.mark.model(dipper.processed,dipper.ddl,
  parameters=list(Phi=PhiFlood, p=pdot))
```

`make.time.factor` *Make time-varying dummy variables from time-varying factor variable*

Description

Create a new dataframe with time-varying dummy variables from a time-varying factor variable. The time-varying dummy variables are named appropriately to be used as a set of time dependent individual covariates in a parameter specification

Usage

```
make.time.factor(x, var.name, times, intercept = NULL, delete = TRUE)
```

Arguments

`x`	dataframe containing set of factor variables with names composed of var.name prefix and times suffix
`var.name`	prefix for variable names
`times`	numeric suffixes for variable names
`intercept`	the value of the factor variable that will be used for the intercept
`delete`	if TRUE, the orginal time-varying factor variables are removed from the returned dataframe

Details

An example of the var.name and times is var.name="observer", times=1:5. The code expects to find observer1,...,observer5 to be factor variables in x. If there are k unique levels (excluding ".") across the time varying factor variables, then k-1 dummy variables are created for each of the named factor variables. They are named with var.name, level[i], times[j] concatenated together where level[i] is the name of the facto level i. If there a m times then the new data set will contain m*(k-1) dummy variables. If the factor variable includes any "." values these are ignored because they are used to indicate a missing value that is paired with a missing value in the encounter history. Note that it will create each dummy variable for each factor even if a particular level is not contained within a factor (eg observers 1 to 3 used but only 1 and 2 on occasion 1).

Value

x: a dataframe containing the original data (with time-varying factor variables removed if delete=TRUE) and the time-varying dummy variables added.

Author(s)

Jeff Laake

Examples

```
# see example in weta
```

---

mallard *Mallard nest survival example*

Description

A nest survival data set on mallards. The data set and analysis is described by Rotella et al.(2004).

Format

A data frame with 565 observations on the following 13 variables.

FirstFound the day the nest was first found

LastPresent the last day that chicks were present

LastChecked the last day the nest was checked

Fate the fate of the nest; 0=hatch and 1 depredated

Freq the frequency of nests with this data; always 1 in this example

Robel Robel reading of vegetation thickness

PpnGrass proportion grass in vicinity of nest

Native dummy 0/1 variable; 1 if native vegetation

Planted dummy 0/1 variable; 1 if planted vegetation

Wetland dummy 0/1 variable; 1 if wetland vegetation

Roadside dummy 0/1 variable; 1 if roadside vegetation

AgeFound age of nest in days the day the nest was found

AgeDay1 age of nest at beginning of study

Details

The first 5 fields must be named as they are shown for nest survival models. `Freq` and the remaining fields are optional. See `killdeer` for more description of the nest survival data structure and the use of the special field `AgeDay1`. The habitat variables `Native`,`Planted`,`Wetland`,`Roadside` were originally coded as 0/1 dummy variables to enable easier modelling with MARK. A better alternative in RMark is to create a single variable `habitat` with values of `"Native"`,`"Planted"`, `"Wetland"`, and `"Roadside"`. Then the Hab model in the example below would become:

```
 Hab=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~habitat)), groups="habitat")
```

For this example, that doesn't make a big difference but if you have more than one factor and you want to construct an interaction or you have a factor with many levels, then it is more efficient to use factor variables rather than dummy variables.

Author(s)

Jay Rotella

Source

Rotella, J.J., S. J. Dinsmore, T.L. Shaffer. 2004. Modeling nest-survival data: a comparison of recently developed methods that can be implemented in MARK and SAS. Animal Biodiversity and Conservation 27:187-204.

Examples

```
# Last updated June 2, 2011
# Read in data, which are in a simple text file that
# looks like a MARK input file but (1) with no comments or semicolons and
# (2) with a 1st row that contains column labels
# mallard=read.table("mallard.txt",header=TRUE)

# The mallard data set is also incuded with RMark and can be retrieved with
# data(mallard)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Example of use of RMark for modeling nest survival data -   #
# Mallard nests example                                       #
# The example runs the 9 models that are used in the Nest     #
# Survival chapter of the Gentle Introduction to MARK and that#
# appear in Table 3 (page 198) of                             #
# Rotella, J.J., S. J. Dinsmore, T.L. Shaffer.  2004.         #
# Modeling nest-survival data: a comparison of recently       #
# developed methods that can be implemented in MARK and SAS.  #
```

```
#   Animal Biodiversity and Conservation 27:187-204.          #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

require(RMark)

# Retrieve data
data(mallard)

# Treat dummy variables for habitat types as factors
mallard$Native=factor(mallard$Native)
mallard$Planted=factor(mallard$Planted)
mallard$Wetland=factor(mallard$Wetland)
mallard$Roadside=factor(mallard$Roadside)

# Examine a summary of the dataset
summary(mallard)

# Write a function for evaluating a set of competing models
run.mallard=function()
{
# 1. A model of constant daily survival rate (DSR)
Dot=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~1)))

# 2. DSR varies by habitat type - treats habitats as factors
#  and the output provides S-hats for each habitat type
Hab=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~Native+Planted+Wetland)),
groups=c("Native","Planted","Wetland"))

# 3. DSR varies with vegetation thickness (Robel reading)
# Note: coefficients are estimated using the actual covariate
# values. They are not based on standardized covariate values.
Robel=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~Robel)))

# 4. DSR varies with the amount of native vegetation in the surrounding area
# Note: coefficients are estimated using the actual covariate
# values. They are not based on standardized covariate values.
PpnGr=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~PpnGrass)))

# 5. DSR follows a trend through time
TimeTrend=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~Time)))

# 6. DSR varies with nest age
Age=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~NestAge)))

# 7. DSR varies with nest age & habitat type
AgeHab=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~NestAge+Native+Planted+Wetland)),
```

```
groups=c("Native","Planted","Wetland"))

# 8. DSR varies with nest age & vegetation thickness
AgeRobel=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~NestAge+Robel)))

# 9. DSR varies with nest age & amount of native vegetation in surrounding area
AgePpnGrass=mark(mallard,nocc=90,model="Nest",
model.parameters=list(S=list(formula=~NestAge+PpnGrass)))

#
# Return model table and list of models
#
return(collect.models() )
}

# The next line runs the 9 models above and takes a minute or 2
mallard.results=run.mallard()
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine table of model-selection results #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

export.MARK(mallard.results$Age$data,"MallDSR",mallard.results,replace=TRUE,ind.covariates="all")
mallard.results                       # print model-selection table to screen
options(width=100)                    # set page width to 100 characters
sink("results.table.txt")             # capture screen output to file
print.marklist(mallard.results)       # send output to file
sink()                                # return output to screen

# remove "#" on next line to see output in notepad
# system("notepad results.table.txt",invisible=FALSE,wait=FALSE)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine output for constant DSR model #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Remove "#" on next line to see output
# mallard.results$Dot                 # print MARK output to designated text editor
mallard.results$Dot$results$beta      # view estimated beta for model in R
mallard.results$Dot$results$real      # view estimated DSR estimate in R

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine output for 'DSR by habitat' model #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Remove "#" on next line to see output
# mallard.results$Hab                 # print MARK output to designated text editor
mallard.results$Hab$design.matrix     # view the design matrix that was used
mallard.results$Hab$results$beta      # view estimated beta for model in R
mallard.results$Hab$results$beta.vcv  # view variance-covariance matrix for beta's
mallard.results$Hab$results$real      # view the estimates of Daily Survival Rate

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Examine output for best model #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
```

```
# Remove "#" on next line to see output
# mallard.results$AgePpnGrass            # print MARK output to designated text editor
mallard.results$AgePpnGrass$results$beta # view estimated beta's in R
mallard.results$AgePpnGrass$results$beta.vcv # view estimated var-cov matrix in R

# To obtain estimates of DSR for various values of 'NestAge' and 'PpnGrass'
#   some work additional work is needed.

# Store model results in object with simpler name
AgePpnGrass=mallard.results$AgePpnGrass
# Build design matrix with ages and ppn grass values of interest
# Relevant ages are age 1 to 35 for mallards
# For ppngrass, use a value of 0.5
fc=find.covariates(AgePpnGrass,mallard)
fc$value[1:35]=1:35                    # assign 1:35 to 1st 35 nest ages
fc$value[fc$var=="PpnGrass"]=0.1       # assign new value to PpnGrass
design=fill.covariates(AgePpnGrass,fc) # fill design matrix with values
# extract 1st 35 rows of output
AgePpn.survival=compute.real(AgePpnGrass,design=design)[1:35,]
# insert covariate columns
AgePpn.survival=cbind(design[1:35,c(2:3)],AgePpn.survival)
colnames(AgePpn.survival)=c("Age","PpnGrass","DSR","seDSR","lclDSR","uclDSR")
AgePpn.survival       # view estimates of DSR for each age and PpnGrass combo

# Plot results
with(data=AgePpn.survival,plot(Age,DSR,'l',ylim=c(0.88,1)))
grid()
axis.break(axis=2,breakpos=0.879,style='slash')
with(data=AgePpn.survival,points(Age,lclDSR,'l',lty=3))
with(data=AgePpn.survival,points(Age,uclDSR,'l',lty=3))

# assign 17 to 1st 50 nest ages
fc$value[1:89]=17
# assign range of values to PpnGrass
fc$value[fc$var=="PpnGrass"]=seq(0.01,0.99,length=89)
design=fill.covariates(AgePpnGrass,fc) # fill design matrix with values
AgePpn.survival=compute.real(AgePpnGrass,design=design)
# insert covariate columns
AgePpn.survival=cbind(design[,c(2:3)],AgePpn.survival)
colnames(AgePpn.survival)=
c("Age","PpnGrass","DSR","seDSR","lclDSR","uclDSR")
# view estimates of DSR for each age and PpnGrass combo
AgePpn.survival

# Plot results
# open new graphics window for new plot
dev.new()
with(data=AgePpn.survival,plot(PpnGrass,DSR,'l',ylim=c(0.88,1)))
grid()
axis.break(axis=2,breakpos=0.879,style='slash')
with(data=AgePpn.survival,points(PpnGrass,lclDSR,'l',lty=3))
with(data=AgePpn.survival,points(PpnGrass,uclDSR,'l',lty=3))
```

```
# The "rm" command can be used to remove all objects from the .Rdata file.
# Cleaning up objects as shown in this script is good programming
# practice and a good idea as long as the computations are not time consuming.
# However, if your analysis is taking several hours or days in MARK then
# clearly you'll want to hang onto the resulting objects in R and you
# won't want to use the following command. It has been commented out for
# this example; the "#" on the next line needs to be removed to do the clean up.
# rm(list=ls(all=TRUE))

# The next line deletes orphaned output files from MARK.
# ?cleanup will give a more complete description of this function.
cleanup(ask=FALSE)
```

`mark` *Interface to MARK for fitting capture-recapture models*

Description

Fits user specified models to various types of capture-recapture data by creating input file and running MARK software and retrieving output

Usage

```
mark(data, ddl = NULL, begin.time = 1, model.name = NULL, model = "CJS",
  title = "", model.parameters = list(), initial = NULL,
  design.parameters = list(), right = TRUE, groups = NULL,
  age.var = NULL, initial.ages = 0, age.unit = 1, time.intervals = NULL,
  nocc = NULL, output = TRUE, invisible = TRUE, adjust = TRUE,
  mixtures = 1, se = FALSE, filename = NULL, prefix = "mark",
  default.fixed = TRUE, silent = FALSE, retry = 0, options = NULL,
  brief = FALSE, realvcv = FALSE, delete = FALSE, external = FALSE,
  profile.int = FALSE, chat = NULL, reverse = FALSE, run = TRUE,
  input.links = NULL, parm.specific = FALSE, mlogit0 = FALSE,
  threads = -1, hessian = FALSE, accumulate = TRUE, allgroups = FALSE,
  strata.labels = NULL, counts = 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.name`	Optional name for the model
`model`	Type of c-r model (eg CJS, Burnham, Barker)
`title`	Optional title for the MARK analysis output

`model.parameters`
: List of model parameter specifications

`initial`
: Optional vector of named or unnamed initial values for beta parameters or previously run model object

`design.parameters`
: Specification of any grouping variables for design data for each parameter

`right`
: if TRUE, any intervals created in design.parameters are closed on the right and open on left and vice-versa if FALSE

`groups`
: Vector of names factor variables for creating groups

`age.var`
: Optional index in groups vector of a variable that represents age

`initial.ages`
: Optional vector of initial ages for each age level

`age.unit`
: Increment of age for each increment of time

`time.intervals`
: Intervals of time between the capture occasions

`nocc`
: number of occasions for Nest model; either time.intervals or nocc must be specified for this model

`output`
: If TRUE produces summary of model input and model output

`invisible`
: If TRUE, window for running MARK is hidden

`adjust`
: If TRUE, adjusts npar to number of cols in design matrix, modifies AIC and records both

`mixtures`
: number of mixtures for heterogeneity model or number of secondary samples for MultScaleOcc model

`se`
: if TRUE, se and confidence intervals are shown in summary sent to screen

`filename`
: base filename for files created by MARK.EXE. Files are named filename.*.

`prefix`
: base filename prefix for files created by MARK.EXE; for example if prefix="SpeciesZ" files are named "SpeciesZnnn.*"

`default.fixed`
: if TRUE, real parameters for which the design data have been deleted are fixed to default values

`silent`
: if TRUE, errors that are encountered are suppressed

`retry`
: number of reanalyses to perform with new starting values when one or more parameters are singular

`options`
: character string of options for Proc Estimate statement in MARK .inp file

`brief`
: if TRUE and output=TRUE then a brief summary line is given instead of a full summary for the model

`realvcv`
: if TRUE the vcv matrix of the real parameters is extracted and stored in the model results

`delete`
: if TRUE the output files are deleted after the results are extracted

`external`
: if TRUE the mark object is saved externally rather than in the workspace; the filename is kept in its place

`profile.int`
: if TRUE will compute profile intervals for each real parameter; or you can specify a vector of real parameter indices

`chat`
: value of chat used for profile intervals

reverse	if set to TRUE, will reverse timing of transition (Psi) and survival (S) in Multi-stratum models
run	if FALSE does not run model after creation
input.links	specifies set of link functions for parameters with non-simplified structure
parm.specific	if TRUE, forces a link to be specified for each parameter
mlogit0	if TRUE, any real parameter that is fixed to 0 and has an mlogit link will have its link changed to logit so it can be simplified
threads	number of cpus to use with mark.exe if positive or number of cpus to remain idle if negative
hessian	if TRUE specifies to MARK to use hessian rather than second partial matrix
accumulate	if TRUE accumulate like data values into frequencies
allgroups	Logical variable; if TRUE, all groups are created from factors defined in groups even if there are no observations in the group
strata.labels	vector of single character values used in capture history(ch) for ORDMS, CRDMS, RDMSOccRepro models; it can contain one more value beyond what is in ch for an unobservable state except for RDMSOccRepro which is used to specify strata ordering (eg 0 not-occupied, 1 occupied no repro, 2 occupied with repro.
counts	named list of numeric vectors (one group) or matrices (>1 group) containing counts for mark-resight models

Details

This function acts as an interface to the FORTRAN program MARK written by Gary White (http://www.cnr.colostate.edu/~gwhite/mark/mark.htm). It creates the input file for MARK based on user-specified sub-models (model.parameters) for each of the parameters in the capture-recapture model being fitted to the data. It runs MARK.EXE (see note below) and then imports the text output file and binary variance-covariance file that were created. It extracts output values from the text file and creates a list of results that is returned as part of the list (of class mark) which is the return value for this function.

The models that are currently supported are listed in MarkModels.pdf which you can find in the RMark sub-directory of your R Library. Also, they are listed under Help/Data Types in the MARK interface.

The function mark is a shell that calls 5 other functions in the following order as needed: 1) process.data, 2) make.design.data, 3) make.mark.model, 4) run.mark.model, and 5) summary.mark. A MARK model can be fitted with this function (mark) or by calling the individual functions that it uses. The calling arguments for mark are a compilation of the calling arguments for each of the functions it calls (with some arguments renamed to avoid conflicts). If data is a processed dataframe (see process.data) then it 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.

For descriptions of the arguments data, begin.time, groups, age.var, initial.ages, age.unit, time.intervals and mixtures see process.data.

For descriptions of ddl, design.parameters=parameters, and right, see make.design.data.

For descriptions of model.name , model, title,model.parameters=parameters , default.fixed , initial, options, see make.mark.model.

And finally, for descriptions of arguments invisible, filename and adjust,see run.mark.model.

output,silent, and retry are the only arguments specific to mark. output controls whether a summary of the model input and output are given(if output=TRUE). silent controls whether errors are shown when fitting a model. retry controls the number of times a model will be refitted with new starting values (uses 0) when some parameters are determined to be non-estimable or at a boundary. The latter is the only time it makes sense to retry with new starting values but MARK cannot discern between these two instances. The indices of the beta parameters that are "singular" are stored in results$singular.

Value

model: a MARK object containing output and extracted results. It is a list with the following elements

data	name of the processed data frame
model	type of analysis model (see list above)
title	title used for analysis
model.name	descriptive name of model
links	vector of link function(s) used for parameters, one for each row in design matrix or only one if all parameters use the same function
mixtures	number of mixtures in Pledger-style closed capture-recapture models
call	call to make.mark.model used to construct the model
parameters	a list of parameter descriptions including the formula, pim.type, link etc.
model.parameters	the list of parameter descriptions used in the call to mark; this is used only by rerun.mark
time.intervals	Intervals of time between the capture occasions
number.of.groups	number of groups defined in the data
group.labels	vector of labels for the groups
nocc	number of capture occasions
begin.time	single time of vector of times (if different for groups) for the first capture occasion
covariates	vector of covariate names (as strings) used in the model
fixed	dataframe of parameters set at fixed values; index is the parameter index in the full parameter structure and value is the fixed value for the real parameter
design.matrix	design matrix used in the input to MARK.EXE
pims	list of pims used for each parameter including any group or strata designations; each parameter in list is denoted by name and within each parameter one or more sub-lists represent groups and strata if any
design.data	design data used to construct the design matrix

strata.labels	labels for strata if any
mlogit.list	structure used to simplify parameters that use mlogit links
simplify	list containing pim.translation which translate between all different and simplified pims, real.labels which are labels for real parameters for full (non-simplified) pim structure and links the link function names for the full parameter structure
output	base portion of filenames for input,output, vc and residual files output from MARK.EXE
results	List of values extracted from MARK ouput

lnl	-2xLog Likelihood value
npar	Number of parameters (always the number of columns in design matrix)
npar.unadjusted	number of estimated parameters from MARK if different than npar
n	effective sample size
AICc	Small sample corrected AIC using npar
AICc.unadjusted	Small sample corrected AIC using npar.unadjusted
beta	data frame of beta parameters with estimate, se, lcl, ucl
real	data frame of real parameters with estimate, se, lcl, ucl and fixed
beta.vcv	variance-covariance matrix for beta
derived	dataframe of derived parameters if any
derived.vcv	variance-covariance matrix for derived parameters if any
covariate.values	dataframe with fields Variable and Value which are the covariate names and value used for real parameter estimates in the MARK output
singular	indices of beta parameters that are non-estimable or at a boundary
real.vcv	variance-covariance matrix for real parameters (simplified) if realvcv=TRUE

chat	over-dispersion constant; if not present assumed to be 1

Note

It is assumed that MARK.EXE is located in directory "C:/Program Files/Mark". If it is in a different location set the variable MarkPath to the directory location. For example, seting MarkPath="C:/Mark/" at the R prompt will assign run "c:/mark/mark.exe" to do the analysis. If you have chosen a non-default path for Mark.exe, MarkPath needs to be defined for each R session. It is easiest to do this assignment automatically by putting the MarkPath assignment into your .First function which is run each time an R session is initiated. In addition to MarkPath, the variable MarkViewer can be assigned to a program other than notepad.exe (see print.mark).

Author(s)

Jeff Laake

See Also

make.mark.model, run.mark.model, make.design.data, process.data, summary.mark

Examples

```
data(dipper)
dipper.Phidot.pdot=mark(dipper)
```

`mark.wrapper` *Constructs and runs a set of MARK models from a dataframe of parameter specifications*

Description

This is a convenience function that uses a dataframe of parameter specifications created by `create.model.list` and it constructs and runs each model and names the models by concatenating each of the parameter specification names separated by a period. The results are returned as a marklist with a model.table constructed by `collect.models`.

Usage

```
mark.wrapper(model.list, silent = FALSE, run = TRUE, use.initial = FALSE,
  initial = NULL, ...)
```

Arguments

`model.list`	a dataframe of parameter specification names in the calling frame
`silent`	if TRUE, errors that are encountered are suppressed
`run`	if FALSE, only calls `make.mark.model` to test for model errors but does not run the models
`use.initial`	if TRUE, initial values are constructed for new models using completed models that have already been run in the set
`initial`	vector, mark model or marklist for defining initial values
`...`	arguments to be passed to `mark`. These must be specified as argument=value pairs.

Details

The model names in `model.list` must be in the frame of the function that calls `run.models`. If `model.list=NULL` or the MARK models are collected from the frame of the calling function (the parent). If `type` is specified only the models of that type (e.g., "CJS") are run. In each case the models are run and saved in the parent frame. To fully understand, how this function works and its limitations, see `create.model.list`.

If `use.initial=TRUE`, prior to running a model it looks for the first model that has already been run (if any) for each parameter formula and constructs an `initial` vector from that previous run. For example, if you provided 5 models for p and 3 for Phi in a CJS model, as soon as the first model for p is run, in the subsequent 2 models with different Phi models, the initial values for p are assigned based on the run with the first Phi model. At the outset this seemed like a good idea to speed up execution times, but from the one set of examples I ran where several parameters were at

boundaries, the results were discouraging because the models converged to a sub-optimal likelihood value than the runs using the default initial values. I've left sthis option in but set its default value to FALSE.

A possibly more useful argument is the argument `initial`. Previously, you could use `initial=model` as part of the ... arguments and it would use the estimates from that model to assign initial values for any model in the set. Now I've defined `initial` as a specific argument and it can be used as above or you can also use it to specify a `marklist` of previously run models. When you do that, the code will lookup each new model to be run in the set of models specified by `initial` and if it finds one with the matching name then it will use the estimates for any matching parameters as initial values in the same way as `initial=model` does. The model name is based on concatenating the names of each of the parameter specification objects. To make this useful, you'll want to adapt to an approach that I've started to use of naming the objects something like p.1,p.2 etc rather than naming them something like p.dot, p.time as done in many of the examples. I've found that using numeric approach is much less typing and cumbersome rather than trying to reflect the formula in the name. By default, the formula is shown in the model selection results table, so it was a bit redundant. Now where I see this being the most benefit. Individual covariate models tend to run rather slowly. So one approach is to run the sequence of models (eg results stored in initial_marklist), including the set of formulas with all of the variables other than individual covariates. Then run another set with the same numbering scheme, but adding the individual covariates to the formula and using `initial=initial_marklist` That will work if each parameter specification has the same name (eg., p.1=list(formula=~time) and then p.1=list(formula=~time+an_indiv_covariate)). All of the initial values will be assigned for the previous run except for any added parameters (eg. an_indiv_covariate) which will start with a 0 initial value.

Value

if(run) marklist - list of mark models that were run and a model.table of results; if(!run) a list of models that were constructed but not run.

Author(s)

Jeff Laake

See Also

`collect.models`, `mark`, `create.model.list`

`mark.wrapper.parallel` *Constructs and runs in parallel a set of MARK models from a dataframe of parameter specifications*

Description

This is a convenience function that uses a dataframe of parameter specifications created by `create.model.list` and it constructs and runs each model and names the models by concatenating each of the parameter specification names separated by a period. The results are returned as a marklist with a model.table constructed by `collect.models`.

Usage

```
mark.wrapper.parallel(model.list, silent = FALSE, use.initial = FALSE,
  initial = NULL, parallel = TRUE, cpus = 2, threads = 1, ...)
```

Arguments

model.list	a dataframe of parameter specification names in the calling frame
silent	if TRUE, errors that are encountered are suppressed
use.initial	if TRUE, initial values are constructed for new models using completed models that have already been run in the set
initial	vector, mark model or marklist for defining initial values
parallel	if TRUE, runs models in parallel on multiple cpus
cpus	number of cpus to use in parallel
threads	number of cpus to use with mark.exe if positive or number of cpus to remain idle if negative
...	arguments to be passed to mark. These must be specified as argument=value pairs.

Details

The model names in model.list must be in the frame of the function that calls run.models. If model.list=NULL or the MARK models are collected from the frame of the calling function (the parent). If type is specified only the models of that type (e.g., "CJS") are run. In each case the models are run and saved in the parent frame. To fully understand, how this function works and its limitations, see create.model.list.

If use.initial=TRUE, prior to running a model it looks for the first model that has already been run (if any) for each parameter formula and constructs an initial vector from that previous run. For example, if you provided 5 models for p and 3 for Phi in a CJS model, as soon as the first model for p is run, in the subsequent 2 models with different Phi models, the initial values for p are assigned based on the run with the first Phi model. At the outset this seemed like a good idea to speed up execution times, but from the one set of examples I ran where several parameters were at boundaries, the results were discouraging because the models converged to a sub-optimal likelihood value than the runs using the default initial values. I've left sthis option in but set its default value to FALSE.

A possibly more useful argument is the argument initial. Previously, you could use initial=model as part of the ... arguments and it would use the estimates from that model to assign initial values for any model in the set. Now I've defined initial as a specific argument and it can be used as above or you can also use it to specify a marklist of previously run models. When you do that, the code will lookup each new model to be run in the set of models specified by initial and if it finds one with the matching name then it will use the estimates for any matching parameters as initial values in the same way as initial=model does. The model name is based on concatenating the names of each of the parameter specification objects. To make this useful, you'll want to adapt to an approach that I've started to use of naming the objects something like p.1,p.2 etc rather than naming them something like p.dot, p.time as done in many of the examples. I've found that using numeric approach is much less typing and cumbersome rather than trying to reflect the formula in

the name. By default, the formula is shown in the model selection results table, so it was a bit redundant. Now where I see this being the most benefit. Individual covariate models tend to run rather slowly. So one approach is to run the sequence of models (eg results stored in initial_marklist), including the set of formulas with all of the variables other than individual covariates. Then run another set with the same numbering scheme, but adding the individual covariates to the formula and using `initial=initial_marklist` That will work if each parameter specification has the same name (eg., p.1=list(formula=~time) and then p.1=list(formula=~time+an_indiv_covariate)). All of the initial values will be assigned for the previous run except for any added parameters (eg. an_indiv_covariate) which will start with a 0 initial value.

Value

marklist - list of mark models that were run and a model.table of results

Author(s)

Eldar Rakhimberdiev

See Also

collect.models, mark, create.model.list

Examples

```
## Not run:
do.MSOccupancy=function()
{
#  Get the data
data(NicholsMSOccupancy)
# Define the models; default of Psi1=~1 and Psi2=~1 is assumed
# p varies by time but p1t=p2t
p1.p2equal.by.time=list(formula=~time,share=TRUE)
# time-varying p1t and p2t
p1.p2.different.time=list(p1=list(formula=~time,share=FALSE),p2=list(formula=~time))
#  delta2 model with one rate for times 1-2 and another for times 3-5;
# delta2 defined below
Delta.delta2=list(formula=~delta2)
Delta.dot=list(formula=~1)  # constant delta
Delta.time=list(formula=~time) # time-varying delta
# Process the data for the MSOccupancy model
NicholsMS.proc=process.data(NicholsMSOccupancy,model="MSOccupancy")
# Create the default design data
NicholsMS.ddl=make.design.data(NicholsMS.proc)
# Add a field for the Delta design data called delta2.  It is a factor variable
# with 2 levels: times 1-2, and times 3-5.
NicholsMS.ddl=add.design.data(NicholsMS.proc,NicholsMS.ddl,"Delta",
type="time",bins=c(0,2,5),name="delta2")
# Create a list using the 4 p modls and 3 delta models (12 models total)
cml=create.model.list("MSOccupancy")
# Fit each model in the list and return the results
return(mark.wrapper.parallel(cml,data=NicholsMS.proc,ddl=NicholsMS.ddl,
    cpus=2,parallel=TRUE))
```

```
}
xx=do.MSOccupancy()

## End(Not run)
```

`mata.wald` *Model-Averaged Tail Area Wald (MATA-Wald) confidence intervals*

Description

A generic function to compute model averaged estimates and their standard errors or variance-covariance matrix model-averaged tail area (MATA) construction.

Usage

```
mata.wald(theta.hats, se.theta.hats, model.weights, normal.lm=FALSE,
                          residual.dfs=0, alpha=0.025)

       tailarea.z(theta, theta.hats, se.theta.hats, model.weights, alpha)

     tailarea.t(theta, theta.hats, se.theta.hats, model.weights, alpha, residual.dfs)
```

Arguments

`theta.hats`	A vector containing the estimates of theta under each candidate model.
`se.theta.hats`	A vector containing the estimated standard error of each estimate in 'theta.hats'.
`model.weights`	A vector containing the model weights for each candidate model. Calculated from an information criterion, such as AIC. See Turek and Fletcher (2012) for details of calculation. All model weights must be non-negative, and sum to one.
`normal.lm`	Specify normal.lm=TRUE for the normal linear model case, and normal.lm=FALSE otherwise. When normal.lm=TRUE, the argument 'residual.dfs' must also be supplied. See USAGE section, and Turek and Fletcher (2012) for additional details.
`residual.dfs`	A vector containing the residual (error) degrees of freedom under each candidate model. This argument must be provided when the argument normal.lm=TRUE.
`alpha`	The desired lower and upper error rate. Specifying alpha=0.025 corresponds to a 95 alpha=0.05 to a 90 Default value is alpha=0.025.
`theta`	value for root finding in tailarea.z and tailarea.t

Details

The main function, mata.wald(...), may be used to construct model-averaged confidence intervals, using the model-averaged tail area (MATA) construction. The idea underlying this construction is similar to that of a model-averaged Bayesian credible interval. This function returns the lower and upper confidence limits of a MATA-Wald interval.

Two usages are supported. For the normal linear model case, and quantity of interest theta, > mata.wald(theta.hats, se.theta.hats, model.weights, alpha, normal.lm=TRUE, residual.dfs) returns a (1-2*alpha)100 Corresponds to the solutions of equations (2) and (3) of Turek and Fletcher (2012). The argument 'residual.dfs' is required for this usage.

When the sampling distribution for the estimate of theta is asymptotically normal (e.g. MLEs), possibly after a transformation, > mata.wald(theta.hats, se.theta.hats, model.weights, alpha, normal.lm=FALSE) returns a (1-2*alpha)100 on a transformed scale. Back-transformation of both confidence limits may be necessary. Corresponds to solutions to the equations in Section 3.2 of Turek and Fletcher (2012).

Author(s)

Daniel Turek<danielturek@gmail.com>

References

Turek, D. and Fletcher, D. (2012). Model-Averaged Wald Confidence Intervals. Computational Statistics and Data Analysis, 56(9), p.2809-2815.

Examples

```
# The example code below, uncommented, generates single-model Wald
# and model-averaged MATA-Wald 95% confidence intervals for theta.
#
#    EXAMPLE: Normal linear prediction
#    =================================
#
# Data 'y', covariates 'x1' and 'x2', all vectors of length 'n'.
# 'y' taken to have a normal distribution.
# 'x1' specifies treatment/group (factor).
# 'x2' a continuous covariate.
#
# Take the quantity of interest (theta) as the predicted response
# (expectation of y) when x1=1 (second group/treatment), and x2=15.

n = 20                              # 'n' is assumed to be even
x1 = c(rep(0,n/2), rep(1,n/2))      # two groups: x1=0, and x1=1
x2 = rnorm(n, mean=10, sd=3)
y = rnorm(n, mean = 3*x1 + 0.1*x2)  # data generation

x1 = factor(x1)
m1 = glm(y ~ x1)                    # using 'glm' provides AIC values.
m2 = glm(y ~ x1 + x2)               # using 'lm' doesn't.
aic = c(m1$aic, m2$aic)
delta.aic = aic - min(aic)
model.weights = exp(-0.5*delta.aic) / sum(exp(-0.5*delta.aic))
residual.dfs = c(m1$df.residual, m2$df.residual)

p1 = predict(m1, se=TRUE, newdata=list(x1=factor(1), x2=15))
p2 = predict(m2, se=TRUE, newdata=list(x1=factor(1), x2=15))
theta.hats = c(p1$fit, p2$fit)
```

```
se.theta.hats = c(p1$se.fit, p2$se.fit)

#  AIC model weights
model.weights

#  95% Wald confidence interval for theta (under Model 1)
theta.hats[1] + c(-1,1)*qt(0.975, residual.dfs[1])*se.theta.hats[1]

#  95% Wald confidence interval for theta (under Model 2)
theta.hats[2] + c(-1,1)*qt(0.975, residual.dfs[2])*se.theta.hats[2]

#  95% MATA-Wald confidence interval for theta (model-averaging)
mata.wald(theta.hats=theta.hats, se.theta.hats=se.theta.hats,
        model.weights=model.weights, normal.lm=TRUE, residual.dfs=residual.dfs)
```

`merge.mark` *Merge mark model objects and lists of mark model objects*

Description

Merge an unspecified number of marklist and mark model objects into a single marklist with an optional table of model results if `table=TRUE`.

Usage

```
## S3 method for class 'mark'
merge(...,table=TRUE)
```

Arguments

`...`	an unspecified number of marklist and/or mark model objects
`table`	if TRUE, a table of model results is also included in the returned list

Value

model.list: a list of `mark` models and optionally a table of model results.

Author(s)

Jeff Laake

See Also

`collect.models`,`remove.mark`,`run.models`,`model.table`,`dipper`

Examples

```
# see example in dipper
```

`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); see example below
`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, add.design.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)

#
# Assign group-specific values
#
dipper.proc=process.data(dipper,groups="sex")
dipper.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(dipper.ddl$p,df,bygroup=TRUE)
```

---

model.average *Compute model averaged estimates*

Description

A generic function to compute model averaged estimates and their standard errors or variance-covariance matrix.

Usage

```
model.average(x, ...)
```

Arguments

x	is either a list with a prescribed structure as defined in model.average.list or a marklist as described in model.average.marklist
...	additional arguments passed to specific functions

Value

The structure of the returned value depends on which function is called.

Author(s)

Jeff Laake

See Also

model.average.marklist,model.average.list

`model.average.list` *Compute model averaged estimates of real parameters from a list structure for estimates*

Description

A generic function to compute model averaged estimates and their standard errors or variance-covariance matrix

Usage

```
## S3 method for class 'list'
model.average(x, revised=TRUE, mata=FALSE, normal.lm=FALSE,
                                        residual.dfs=0, alpha=0.025,...)
```

Arguments

`x`	a list containing the following elements: 1) `estimate` - a vector or matrix of estimates, 2)a vector of model selection criterion value named `AIC,AICc,QAIC,QAICc` or a `weight` variable that sums to 1 across models, and 3) a vector or matrix named `se` which give the model-specific standard errors for each estimate or a list of matrices named `vcv` which give the model-specific variance-covariance matrices.
`revised`	if TRUE it uses eq 6.12 from Burnham and Anderson (2002) for model averaged se; otherwise it uses eq 4.9
`mata`	if TRUE, create model averaged tail area confidence intervals as described by Turek and Fletcher
`normal.lm`	Specify normal.lm=TRUE for the normal linear model case, and normal.lm=FALSE otherwise. When normal.lm=TRUE, the argument 'residual.dfs' must also be supplied. See USAGE section, and Turek and Fletcher (2012) for additional details.
`residual.dfs`	A vector containing the residual (error) degrees of freedom under each candidate model. This argument must be provided when the argument normal.lm=TRUE.
`alpha`	The desired lower and upper error rate. Specifying alpha=0.025 corresponds to a 95 alpha=0.05 to a 90 Default value is alpha=0.025.
`...`	additional arguments passed to specific functions

Details

If a single estimate is being model-averaged then `estimate` and `se` are vectors with an entry for each model. However, if there are several estimatee being averaged then both `estimate` and `se` should be matrices in which the estimates for each model are a row in the matrix. Regardless, if `vcv` is specified it should be a list of matrices and in the case of a single estimate, each matrix is 1x1 containing the estimated sample-variance but that would be rather useless and `se` should be used instead.

If the list contains an element named `AIC`,`AICc`,`QAIC`, or `QAICc`, then the minimum value is computed and subtracted to compute delta values relative to the minimum. These are then converted to Akaike weights which are `exp(-.5*delta)` and these are normalized to sum to 1. If the list does not contain one of the above values then it should have a variable named `weight`. It is normalized to 1. The model-averaged estimates are computed using equation 4.1 of Burnham and Anderson (2002).

If the contains a matrix named `vcv`, then a model-averaged variance-covariance matrix is computed using formulae given on page 163 of Burnham and Anderson (2002). If there is no element named `vcv` then there must be an element `se` which contains the model-specific estimates of the standard errors. The unconditional standard error for the model-averaged estimates is computed using equation 4.9 of Burnham and Anderson (2002) if if `revised=FALSE`; otherwise it uses eq 6.12.

Value

A list containing elements:

`estimate`	vector of model-averaged estimates
`se`	vector of unconditional standard errors (square root of unconditional variance estimator)
`vcv`	model-averaged variance-covariance matrix if `vcv` was specified input list
`lcl`	lower confidence interval if mata=TRUE
`ucl`	upper confidence interval if mata=TRUE

Author(s)

Jeff Laake

References

BURNHAM, K. P., AND D. R. ANDERSON. 2002. Model selection and multimodel inference. A practical information-theoretic approach. Springer, New York. Turek, D. and Fletcher, D. (2012). Model-Averaged Wald Confidence Intervals. Computational Statistics and Data Analysis, 56(9), p.2809-2815.

See Also

`model.average.marklist`

Examples

```
# Create a set of models from dipper data
data(dipper)
run.dipper=function()
{
dipper$nsex=as.numeric(dipper$sex)-1
mod1=mark(dipper,groups="sex",
   model.parameters=list(Phi=list(formula=~sex)))
mod2=mark(dipper,groups="sex",
   model.parameters=list(Phi=list(formula=~1)))
mod3=mark(dipper,groups="sex",
   model.parameters=list(p=list(formula=~time),
   Phi=list(formula=~1)))
dipper.list=collect.models()
return(dipper.list)
}
dipper.results=run.dipper()
# Extract indices for first year survival from
#  Females (group 1) and Males (group 2)
Phi.indices=extract.indices(dipper.results[[1]],
   "Phi",df=data.frame(group=c(1,2),row=c(1,1),col=c(1,1)))
# Create a matrix for estimates
estimate=matrix(0,ncol=length(Phi.indices),
    nrow=nrow(dipper.results$model.table))
# Extract weights for models
weight=dipper.results$model.table$weight
# Create an empty list for vcv matrices
vcv=vector("list",length=nrow(dipper.results$model.table))
# Loop over each model in model.table for dipper.results
for (i in 1:nrow(dipper.results$model.table))
{
# The actual model number is the row number for the model.table
  model.numbers= as.numeric(row.names(dipper.results$model.table))
# For each model extract those real parameter values and their
# vcv matrix and store them
  x=covariate.predictions(dipper.results[[model.numbers[i]]],
      data=data.frame(index=Phi.indices))
  estimate[i,]=x$estimates$estimate
  vcv[[i]]=x$vcv
}
# Call model.average using the list structure which includes
#  estimate, weight and vcv list in this case
model.average(list(estimate=estimate,weight=weight,vcv=vcv))
#
# Now get same model averaged estimates using model.average.marklist
# Obviously this is a much easier approach and what would be used
# if all you are doing is model averaging real parameters in the model.
# The other form is more useful for model averaging
# functions of estimates of the real parameters (eg population estimate)
#
mavg=model.average(dipper.results,"Phi",vcv=TRUE)
```

```
print(mavg$estimates[Phi.indices,])
print(mavg$vcv.real[Phi.indices,Phi.indices])
```

`model.average.marklist` *Compute model averaged estimates of real parameters*

Description

Computes model averaged estimates and standard errors of real parameters for a list of models with a `model.table` constructed from `collect.models`. It can also optionally compute the var-cov matrix of the averaged parameters and their confidence intervals by transforming with the link functions, setting normal confidence intervals on the transformed values and then back-transforming for the real estimates.

Usage

```
## S3 method for class 'marklist'
model.average(x, parameter, data, vcv, drop=TRUE, indices=NULL, revised=TRUE, mata=FALSE,
        normal.lm=FALSE, residual.dfs=0, alpha=0.025,...)
```

Arguments

`x`	a list of mark model results and a model.table constructed by `collect.models`
`parameter`	name of model parameter (e.g., "Phi" for CJS models); if left NULL all real parameters are averaged
`data`	dataframe with covariate values that are averaged for estimates
`vcv`	logical; if TRUE then the var-cov matrix and confidence intervals are computed
`drop`	if TRUE, models with any non-positive variance for betas are dropped
`indices`	a vector of parameter indices from the all-different PIM formulation of the parameter estimates that should be presented. This argument only works if the parameter argument = NULL. The primary purpose of the argument is to trim the list of parameters in computing a vcv matrix of the real parameters which can get too big to be computed with the available memory
`revised`	if TRUE, uses revised variance formula (eq 6.12 from Burnham and Anderson) for model averaged estimates and eq 6.11 when FALSE
`mata`	if TRUE, create model averaged tail area confidence intervals as described by Turek and Fletcher
`normal.lm`	Specify normal.lm=TRUE for the normal linear model case, and normal.lm=FALSE otherwise. When normal.lm=TRUE, the argument 'residual.dfs' must also be supplied. See USAGE section, and Turek and Fletcher (2012) for additional details.
`residual.dfs`	A vector containing the residual (error) degrees of freedom under each candidate model. This argument must be provided when the argument normal.lm=TRUE.

alpha	The desired lower and upper error rate. Specifying alpha=0.025 corresponds to a 95 alpha=0.05 to a 90 Default value is alpha=0.025.
...	additional arguments passed to specific functions

Details

If there are any models in the model.list which do not have any output or results they are dropped. If any have non-positive variances for the betas and drop=TRUE, then the model is reported and dropped from the model averaging. The weights are renormalized for the remaining models that are not dropped before they are averaged.

If parameter=NULL, all real parameters are model averaged but the design data is not copied over because it can vary by the type of parameter. It is only necessary to model average all parameters at once to get covariances of model averaged parameters of differing types.

If data=NULL, the average covariate values are used for any models using covariates. Note that this will only work with models created after v1.5.0 such that average covariate values are stored in each model object.

Value

If vcv=FALSE, the return value is a dataframe of model averaged estimates and standard errors for a particular type of real parameter (e.g., Phi). The design data are appended to the dataframe to enable subsettting of the estimates based on features of the design data such as age, time, cohort and grouping variables.

If vcv=TRUE, confidence interval (lcl,ucl) limits are added to the dataframe which is contained in a list with the var-cov matrix.

Author(s)

Jeff Laake

References

Burnham, K. P. and D. R. Anderson. 2002. Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach, Second edition. Springer, New York.

See Also

collect.models, covariate.predictions, model.table, compute.links.from.reals, model.average.list

Examples

```
data(dipper)
run.dipper=function()
{
#
# Process data
#
dipper.processed=process.data(dipper,groups=("sex"))
```

```
#
# Create default design data
#
dipper.ddl=make.design.data(dipper.processed)
#
# Add Flood covariates for Phi and p that have different values
#
dipper.ddl$Phi$Flood=0
dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 | dipper.ddl$Phi$time==3]=1
dipper.ddl$p$Flood=0
dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
#
#  Define range of models for Phi
#
Phi.dot=list(formula=~1)
Phi.time=list(formula=~time)
Phi.sex=list(formula=~sex)
Phi.sextime=list(formula=~sex+time)
Phi.sex.time=list(formula=~sex*time)
Phi.Flood=list(formula=~Flood)
#
#  Define range of models for p
#
p.dot=list(formula=~1)
p.time=list(formula=~time)
p.sex=list(formula=~sex)
p.sextime=list(formula=~sex+time)
p.sex.time=list(formula=~sex*time)
p.Flood=list(formula=~Flood)
#
# Collect pairings of models
#
cml=create.model.list("CJS")
#
# Run and return the list of models
#
return(mark.wrapper(cml,data=dipper.processed,ddl=dipper.ddl))
}
dipper.results=run.dipper()
Phi.estimates=model.average(dipper.results,"Phi",vcv=TRUE)
p.estimates=model.average(dipper.results,"p",vcv=TRUE)
run.dipper=function()
{
data(dipper)
dipper$nsex=as.numeric(dipper$sex)-1
#NOTE:  This generates random values for the weights so the answers using
#  ~weight will vary
dipper$weight=rnorm(294)
mod1=mark(dipper,groups="sex",
  model.parameters=list(Phi=list(formula=~sex+weight)))
mod2=mark(dipper,groups="sex",
  model.parameters=list(Phi=list(formula=~sex)))
mod3=mark(dipper,groups="sex",
```

```
  model.parameters=list(Phi=list(formula=~weight)))
mod4=mark(dipper,groups="sex",
  model.parameters=list(Phi=list(formula=~1)))
dipper.list=collect.models()
return(dipper.list)
}
dipper.results=run.dipper()
real.averages=model.average(dipper.results,vcv=TRUE)
# get model averaged estimates for all parameters and use average
# covariate values in models with covariates
real.averages$estimates
# get model averaged estimates for Phi using a value of 2 for weight
model.average(dipper.results,"Phi",
  data=data.frame(weight=2),vcv=FALSE)
# what you can't do yet is use different covariate values for
# different groups to get covariances of estimates based on different
# covariate values; for example, you can get average survival of females
# at average female weight and average survival of males at average
# male weight in separate calls to model.average but not in the same call
# to get covariances; however, if you standardized weight by group
# (ie stdwt = weight - groupmean) then using 0 for the covariate value would give
# the model averaged Phi by group at the average group weights and its
# covariance. You can do the above for
# a single model with find.covariates/fill.covariates.
# get model averaged estimates of first Phi(1) and first p(43) and v-c matrix
model.average(dipper.results,vcv=TRUE,indices=c(1,43))
```

`model.table` *Create table of MARK model selection results*

Description

Constructs a table of model selection results for MARK analyses. The table includes the formulas, model name, number of parameters, deviance, AICc, DeltaAICc, model weight and residual deviance. If chat>1 QAICc, QDeltaAICc and QDeviance are used instead.

Usage

```
model.table(model.list = NULL, type = NULL, sort = TRUE, adjust = TRUE,
  ignore = TRUE, pf = 1, use.lnl = FALSE, use.AIC = FALSE,
  model.name = TRUE)
```

Arguments

`model.list`	a vector of model names or a list created by the function `collect.models` which has each model object and at the end a `model.table` ; If nothing is specified then any mark object in the workspace is collected for the table. If `type` is specified all analyses in parent frame(`pf`) of that `type` of model are used. If specified set

	of models are of conflicting types or of different data sets then an error is issued unless ignore=TRUE
type	type of model (eg "CJS")
sort	if true sorts models by criterion
adjust	if TRUE adjusts # of parameters to # of cols in design matrix
ignore	if TRUE collects all models and ignores that they are from different models
pf	parent frame value; default=1 so it looks in calling frame of model.table; used in other functions with pf=2 when functions are nested two-deep
use.lnl	display -2lnl instead of deviance
use.AIC	use AIC instead of AICc
model.name	if TRUE uses the model.name in each mark object which uses formula notation. If FALSE it uses the R names for the model obtained from collect.model.names or names assigned to marklist elements

Details

This function is used by collect.models to construct a table of model selection results with the models that it collects; however it can be called directly to construct the table.

Value

result.table - dataframe containing summary of models

model.name	name of fitted model
parameter.name - an entry for each parameter	formula for parameter
npar	number of estimated parameters
AICc or QAICc	AICc value or QAICc if chat>1
DeltaAICc or DeltaQAICc	difference between AICc or QAICc value from model with smallest value
weight	model weight based on exp(-.5*DeltaAICc) or exp(-.5*QDeltaAICc)
Deviance or QDeviance	residual deviance from saturated model
chat	overdispersion constant if not 1

Author(s)

Jeff Laake

See Also

collect.model.names, collect.models

Examples

```
data(dipper)
run.dipper=function()
{
#
# Process data
#
dipper.processed=process.data(dipper,groups=("sex"))
#
# Create default design data
#
dipper.ddl=make.design.data(dipper.processed)
#
# Add Flood covariates for Phi and p that have different values
#
dipper.ddl$Phi$Flood=0
dipper.ddl$Phi$Flood[dipper.ddl$Phi$time==2 | dipper.ddl$Phi$time==3]=1
dipper.ddl$p$Flood=0
dipper.ddl$p$Flood[dipper.ddl$p$time==3]=1
#
#  Define range of models for Phi
#
Phi.dot=list(formula=~1)
Phi.time=list(formula=~time)
Phi.sex=list(formula=~sex)
Phi.sextime=list(formula=~sex+time)
Phi.sex.time=list(formula=~sex*time)
Phi.Flood=list(formula=~Flood)
#
#  Define range of models for p
#
p.dot=list(formula=~1)
p.time=list(formula=~time)
p.sex=list(formula=~sex)
p.sextime=list(formula=~sex+time)
p.sex.time=list(formula=~sex*time)
p.Flood=list(formula=~Flood)
#
# Return model table and list of models
#
cml=create.model.list("CJS")
return(mark.wrapper(cml,data=dipper.processed,ddl=dipper.ddl))
}

dipper.results=run.dipper()
dipper.results
dipper.results$model.table=model.table(dipper.results,model.name=FALSE)
dipper.results
#
# Compute matrices of model weights, number of parameters and Delta AICc values
#
```

```
model.weight.matrix=tapply(dipper.results$model.table$weight,
 list(dipper.results$model.table$Phi,dipper.results$model.table$p),mean)
model.npar.matrix=tapply(dipper.results$model.table$npar,
 list(dipper.results$model.table$Phi,dipper.results$model.table$p),mean)
model.DeltaAICc.matrix=tapply(dipper.results$model.table$DeltaAICc,
 list(dipper.results$model.table$p,dipper.results$model.table$Phi),mean)
#
# Output DeltaAICc as a tab-delimited text file that can be read into Excel
# (to do that directly use RODBC or xlsreadwrite package for R)
#
write.table(model.DeltaAICc.matrix,"DipperDeltaAICc.txt",sep="\t")
```

`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. The models created by RMark are all "Parm-specific" models by default. The sin link is not allowed because all models are specified via the design matrix. Although you can set links for the parameters, usually the default values are preferable. See `make.mark.model` for additional help building formula for Psi using the remove.intercept argument.

Examples

```
data(mstrata)
run.mstrata=function()
{
#
# Process data
#
mstrata.processed=process.data(mstrata,model="Multistrata")
#
# Create default design data
#
```

```
mstrata.ddl=make.design.data(mstrata.processed)
#
#  Define range of models for S; note that the betas will differ from the output
#  in MARK for the ~stratum = S(s) because the design matrix is defined using
#  treatment contrasts for factors so the intercept is stratum A and the other
#  two estimates represent the amount that survival for B abd C differ from A.
#  You can use force the approach used in MARK with the formula ~-1+stratum which
#  creates 3 separate Betas - one for A,B and C.
#
S.stratum=list(formula=~stratum)
S.stratumxtime=list(formula=~stratum*time)
#
#  Define range of models for p
#
p.stratum=list(formula=~stratum)
#
#  Define range of models for Psi; what is denoted as s for Psi
#  in the Mark example for Psi is accomplished by -1+stratum:tostratum which
#  nests tostratum within stratum.  Likewise, to get s*t as noted in MARK you
#  want ~-1+stratum:tostratum:time with time nested in tostratum nested in
#  stratum.
#
Psi.s=list(formula=~-1+stratum:tostratum)
Psi.sxtime=list(formula=~-1+stratum:tostratum:time)
#
# Create model list and run assortment of models
#
model.list=create.model.list("Multistrata")
#
# Add on specific models that are paired with fixed p's to remove confounding
#
p.stratumxtime=list(formula=~stratum*time)
p.stratumxtime.fixed=list(formula=~stratum*time,fixed=list(time=4,value=1))
model.list=rbind(model.list,c(S="S.stratumxtime",p="p.stratumxtime.fixed",
  Psi="Psi.sxtime"))
model.list=rbind(model.list,c(S="S.stratum",p="p.stratumxtime",Psi="Psi.s"))
#
# Run the list of models
#
mstrata.results=mark.wrapper(model.list,data=mstrata.processed,ddl=mstrata.ddl,threads=2)
#
# Return model table and list of models
#
return(mstrata.results)
}
mstrata.results=run.mstrata()
mstrata.results

# Example of reverse Multistratum model
data(mstrata)
mod=mark(mstrata,model="Multistrata")
mod.rev=mark(mstrata,model="Multistrata",reverse=TRUE)
Psilist=get.real(mod,"Psi",vcv=TRUE)
```

```
Psilist.rev=get.real(mod.rev,"Psi",vcv=TRUE)
Psivalues=Psilist$estimates
Psivalues.rev=Psilist.rev$estimates
TransitionMatrix(Psivalues[Psivalues$time==1,])
TransitionMatrix(Psivalues.rev[Psivalues.rev$occ==1,])
```

NicholsMSOccupancy *Multi-state occupancy example data*

Description

An occupancy data set for modelling multi-state data (0,1,2).

Format

A data frame with 40 records for 54 observations (sites) on the following 2 variables.

ch a character vector containing the presence (state 1), presence (state 2), and absence (0) for each visit to the site, and a "." if the site was not visited

freq frequency of sites with that history

Details

This is a data set from Nichols et al (2007).

References

Nichols, J. D., J. E. Hines, D. I. MacKenzie, M. E. Seamans, and R. J. Gutierrez. 2007. Occupancy estimation and modeling with multiple states and state uncertainty. Ecology 88:1395-1400.

Examples

```
# To create the data file use:
# NicholsMSOccupancy=convert.inp("NicholsMSOccupancy.inp")
#
# Create a function to fit the 12 models in Nichols et al (2007).
do.MSOccupancy=function()
{
#  Get the data
   data(NicholsMSOccupancy)
# Define the models; default of Psi1=~1 and Psi2=~1 is assumed
   # p varies by time but p1t=p2t
   p1.p2equal.by.time=list(formula=~time,share=TRUE)
   # time-invariant p p1t=p2t=p1=p2
   p1.p2equal.dot=list(formula=~1,share=TRUE)
   #time-invariant p1 not = p2
   p1.p2.different.dot=list(p1=list(formula=~1,share=FALSE),p2=list(formula=~1))
```

```
   # time-varying p1t and p2t
   p1.p2.different.time=list(p1=list(formula=~time,share=FALSE),p2=list(formula=~time))
   # delta2 model with one rate for times 1-2 and another for times 3-5;
   #delta2 defined below
   Delta.delta2=list(formula=~delta2)
   Delta.dot=list(formula=~1)  # constant delta
   Delta.time=list(formula=~time) # time-varying delta
# Process the data for the MSOccupancy model
   NicholsMS.proc=process.data(NicholsMSOccupancy,model="MSOccupancy")
# Create the default design data
   NicholsMS.ddl=make.design.data(NicholsMS.proc)
# Add a field for the Delta design data called delta2.  It is a factor variable
# with 2 levels: times 1-2, and times 3-5.
   NicholsMS.ddl=add.design.data(NicholsMS.proc,NicholsMS.ddl,"Delta",
     type="time",bins=c(0,2,5),name="delta2")
# Create a list using the 4 p modls and 3 delta models (12 models total)
   cml=create.model.list("MSOccupancy")
# Fit each model in the list and return the results
   return(mark.wrapper(cml,data=NicholsMS.proc,ddl=NicholsMS.ddl))
}
# Call the function to fit the models and store it in MSOccupancy.results
MSOccupancy.results=do.MSOccupancy()
# Print the model table for the results
print(MSOccupancy.results)
# Adjust model selection by setting chat=1.74
MSOccupancy.results=adjust.chat(chat=1.74,MSOccupancy.results)
# Print the adjusted model selection results table
print(MSOccupancy.results)
#
# To fit an additive model whereby p1 and p2 differ by time and p2 differs from
# p1 a constant amount on the logit scale, use
#
# p varies by time logit(p1t)=logit(p2t)+constant
p1.plust.p2.by.time=list(formula=~time+p2,share=TRUE)
```

PIMS *Display PIM for a parameter*

Description

Extract PIMS for a particular parameter and display either the full PIM structure or the simplified PIM structure.

Usage

```
PIMS(model, parameter, simplified = TRUE, use.labels = TRUE)
```

Arguments

model	mark model object
parameter	character string of a particular type of parameter in the model (eg "p","Phi","pent","S")
simplified	if TRUE show simplified PIM structure; otherwise show full structure
use.labels	if TRUE, uses time and cohort labels for columns and rows respectively

Value

None

Author(s)

Jeff Laake

See Also

make.design.data

Examples

```
data(dipper)
results=mark(dipper)
PIMS(results,"Phi")
PIMS(results,"Phi",simplified=FALSE)
```

PoissonMR *Example of Poisson Mark-Resight model*

Description

Data and example illustrating Poisson Mark-Resight model.

Format

A data frame with 68 observations on the following 1 variables.

ch a character vector

Examples

```
data(PoissonMR)
pois.proc=process.data(PoissonMR,model="PoissonMR",
counts=list("Unmarked Seen"=c(1380, 1120, 1041, 948),
    "Marked Unidentified"=c(8,10,9,11),
"Known Marks"=c(45,67,0,0)))
pois.ddl=make.design.data(pois.proc)
mod=mark(pois.proc,pois.ddl,
model.parameters=list(Phi=list(formula=~1,link="sin"),
      GammaDoublePrime=list(formula=~1,share=TRUE,link="sin"),
      alpha=list(formula=~-1+time,link="log"),
  U=list(formula=~-1+time,link="log"),
  sigma=list(formula=~-1+time,link="log")),
         initial=c(1,1,1,1,-1.4,-.8,-.9,-.6,6,6,6,6,2,-1),threads=2)
summary(mod)
```

Poisson_twoMR *Example of Poisson Mark-Resight model*

Description

Data and example illustrating Poisson Mark-Resight model with 2 groups and one occasion.

Format

A data frame with 93 observations on the following 2 variables.

ch a character vector

pg a factor with levels group1 group2

Examples

```
data(Poisson_twoMR)
pois.proc=process.data(Poisson_twoMR,model="PoissonMR",groups="pg",
counts=list("Unmarked Seen"=matrix(c(1237,588),nrow=2,ncol=1),
    "Marked Unidentified"=matrix(c(10,5),nrow=2,ncol=1),
"Known Marks"=matrix(c(60,0),nrow=2,ncol=1)))
pois.ddl=make.design.data(pois.proc)
mod=mark(pois.proc,pois.ddl,
model.parameters=list(alpha=list(formula=~1),
              U=list(formula=~-1+group),
              sigma=list(formula=~1,fixed=0)),
                        initial=c(0.9741405 ,0.0000000 ,6., 5.),threads=2)
summary(mod)
```

popan.derived *Computes some derived abundance estimates for POPAN models*

Description

Computes estimates, standard errors, confidence intervals and var-cov matrix for population size of each group at each occasion and the sum across groups by occasion for POPAN models. If a marklist is provided the estimates are model averaged.

Usage

```
popan.derived(x,model,revised=TRUE,normal=TRUE,N=TRUE,NGross=TRUE,drop=FALSE)

popan.Nt(Phi,pent,Ns,vc,time.intervals)

popan.NGross(Phi,pent,Ns,vc,time.intervals)
```

Arguments

x	processed data list resulting from process.data
model	a single mark POPAN model or a marklist of POPAN models
revised	if TRUE, uses revised version of model averaged standard error eq 6.12; otherwise uses eq 4.9 of Burnham and Anderson (2002)
normal	if TRUE, uses confidence interval based on normal distribution; otherwise, uses log-normal
N	if TRUE, will return abundance estimates by group and occasion and total by occasion
NGross	if TRUE, will return gross abundance estimate per group
drop	if TRUE, models with any non-positive variance for betas are dropped
Phi	interval-specific survival estimates for each group
pent	occasion-specific prob of entry estimates (first computed by subtraction) for each group
Ns	group specific super-population estimate
vc	variance-covariance matrix of the real parameters
time.intervals	vector of time interval values

Details

popan.derived computes all of the real parameters using covariate.predictions and handles all of the computation using popan.Nt. Description for functions popan.Nt and popan.NGross are given here for completeness but it is not intended that they be called directly.

If a model is a marklist of models, the values returned by popan.derived are model averaged using model weights in the model.table; otherwise, it returns the values for the specified model.

Value

popan.derived returns a list with the following elements depending on the values of N and NGross:

```
 N -
dataframe of estimates by group and occasion and se, lcl,ucl and
group/occasion data N.vcv - variance-covariance matrix of abundance
estimates in N Nbyocc - dataframe of estimates by occasion (summed across
groups) and se, lcl,ucl and occasion data Nbyocc.vcv - variance-covariance
matrix of abundance estimates in Nbyocc NGross - dataframe of gross
abundance estimates by group and se, lcl,and ucl NGross.vcv -
variance-covariance matrix of NGross abundance estimates
```

popan.Nt returns a list with the following elements:

```
 N
- dataframe of estimates by group and occasion and se, lcl,ucl and
group/occasion data N.vcv - variance-covariance matrix of abundance
estimates in N
```

popan.NGross returns a list with the following elements:

```
 NGross - vector of gross abundance estimates by group vcv -
variance-covariance matrix of abundance estimates in NGross
```

Author(s)

Jeff Laake

References

BURNHAM, K. P., AND D. R. ANDERSON. 2002. Model selection and multimodel inference. A practical information-theoretic approach. Springer, New York.

Examples

```
# Example
data(dipper)
dipper.processed=process.data(dipper,model="POPAN",groups="sex")
run.dipper.popan=function()
{
dipper.ddl=make.design.data(dipper.processed)
Phidot=list(formula=~1)
Phitime=list(formula=~time)
pdot=list(formula=~1)
ptime=list(formula=~time)
pentsex.time=list(formula=~time)
Nsex=list(formula=~sex)
#
# Run assortment of models
```

```
#
dipper.phisex.time.psex.time.pentsex.time=mark(dipper.processed,
     dipper.ddl,model.parameters=list(Phi=Phidot,p=ptime,
     pent=pentsex.time,N=Nsex),invisible=FALSE,adjust=FALSE)
dipper.psex.time.pentsex.time=mark(dipper.processed,dipper.ddl,
     model.parameters=list(Phi=Phitime,p=pdot,
     pent=pentsex.time,N=Nsex),invisible=FALSE,adjust=FALSE)
#
# Return model table and list of models
#
return(collect.models() )
}
dipper.popan.results=run.dipper.popan()
popan.derived(dipper.processed,dipper.popan.results)
```

`print.mark` *Print MARK objects*

Description

If print is for a mark model, it displays MARK output file or input file with MarkViewer (notepad.exe by default) so it can be viewed. If print is for a marklist, it displays the model.table if it exists. To display the output for a mark model contained in a list, simply type the list value (e.g., typing mymarklist[[2]] will display output for the second model). The function `print.marklist` was created to avoid accidental typing of the model list which would call `print.mark` for each of the models.

Usage

```
## S3 method for class 'mark'
print(x,...,input=FALSE)
       ## S3 method for class 'marklist'
print(x,...)
```

Arguments

`x`	mark model object; or list of mark model objects created with `collect.models`
`...`	additional non-specified argument for S3 generic function
`input`	if TRUE, prints mark input file; otherwise the output file

Details

If the model has been run (`model$output` exists) the output file stored in the directory as identified by the basefile name (`model$output`) and the suffix ".out" is displayed with a call to `MarkViewer`. If `input` is set to `TRUE` then the MARK input file is displayed instead. By default the `MarkViewer` is notepad but any program can be used in its place that accepts the filename as the first argument. For example setting `MarkViewer="wp"` will use wordperfect (wp.exe) as long as wp.exe is in the search path. `MarkViewer` must be set during each R session, so it is best to include it in your `.First`

function to change it permanently. Since `print.mark` is the generic function to print `mark` objects you can use it by just typing the name of a `mark` object at the R prompt and it will call `print.mark`. For example, if `mod` is a `mark` object then typing `mod` is the same as `print.mark(mod)`

Value

None

Author(s)

Jeff Laake

See Also

`summary.mark`

`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 more 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),
  age.unit = 1, time.intervals = NULL, nocc = NULL,
  strata.labels = NULL, counts = NULL, reverse = FALSE)
```

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. See `mark` for a list of possible values for `model`
`mixtures`	Number of mixtures in closed capture models with heterogeneity or number of secondary samples for MultScaleOcc model

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]
age.unit	Increment of age for each increment of time as defined by time.intervals
time.intervals	Vector of lengths of time between capture occasions
nocc	number of occasions for Nest type; either nocc or time.intervals must be specified
strata.labels	vector of single character values used in capture history(ch) for ORDMS, CRDMS, RDMSOccRepro models; it can contain one more value beyond what is in ch for an unobservable state except for RDMSOccRepro which is used to specify strata ordering (eg 0 not-occupied, 1 occupied no repro, 2 occupied with repro.
counts	named list of numeric vectors (one group) or matrices (>1 group) containing counts for mark-resight models
reverse	if set to TRUE, will reverse timing of transition (Psi) and survival (S) in Multistratum models

Details

For examples of data, see dipper,edwards.eberhardt,example.data. 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; however, this is dependent on the type of analysis model. For models such as "CJS", "Pradel" and others, it is simply the length of ch. Whereas, for "Burnham" and "Barker" models,the encounter history contains both capture and resight/recovery values so nocc is one-half the length of ch. Likewise, the number of time.intervals depends on the model. For models, such as "CJS", "Pradel" and others, the number of time.intervals is nocc-1; whereas, for capture&recovery(resight) models the number of time.intervals is nocc. 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, add.design.data, and make.mark.model 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) and dipper.popan=process.data(dipper,model="POPAN").

This function will report inconsistencies in the lengths of the capture history values and when invalid entries are given in the capture history. For example, with the "CJS" model, the capture history should only contain 0 and 1 whereas for "Barker" it can contain 0,1,2. For "Multistrata" models, the code will automatically identify the number of strata and strata labels based on the unique alphabetic codes used in the capture histories.

The argument begin.time specifies the time for the first capture occasion. 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. 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.

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 is 0,1 and 2 while the age classes were 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 argument age.unit is the amount an animal ages for each unit of time and the default is 1. 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.

Value

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", "Burnham", "Barker")
freq	a dataframe of frequencies (same number 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
age.unit	increment of age for each increment of time
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
nstrata	number of strata in Multistrata models
strata.labels	vector of alphabetic characters used to identify strata in Multistrata models
group.covariates	factor covariates used to define groups

Author(s)

Jeff Laake

See Also

import.chdata, dipper, edwards.eberhardt, example.data

Examples

```
data(example.data)
proc.example.data=process.data(data=example.data,begin.time=1980,
groups=c("sex","age","region"),
age.var=2,initial.age=c(0,1,2))

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

RDOccupancy *Robust Design occupancy example data*

Description

A simulated data set on a breeding bird as an example of robust design occupancy modeling.

Format

A data frame with 35 observations on the following 12 variables

ch A character vector containing the presence (1) and absence (0) or (.) not visited for each of 3 visits (secondary occasions) over 3 years (primary occasions)

cover percentage canopy cover at each sampled habitat

occ11 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ12 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ13 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ21 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ22 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ23 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ31 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ32 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

occ33 one of 9 session-dependent variables occ11 to occ33 containing the week the survey was conducted; p is the primary session number and s is the secondary session number

samplearea continuous variable indicating area size (ha) of the sampled habitat

Details

These are simulated data for an imaginary situation with 35 independent 'sites' on which presence/absence of a breeding bird is recorded 3 times annually for 3 years. Potential variables influencing site occupancy are the size of the site in hectares (samplearea) and canopy cover percentage (cover). The timing of the surveys within the year is thought to influence the detection of occupancy, so the week the survey was conducted is included in 9 variables that are named as occps where p is the primary session (year) number and s is the secondary session (visit) number. Using `data(RDOccupancy)` will retrieve the completed dataframe and using `example(RDOccpancy)` will run the example code. However, in this example we also show how to import the raw data and how they were modified to construct the `RDOccupancy` dataframe.

For this example, the raw data are shown below and the code below assumes the file is named `RD_example.txt`.

```
 ch samplearea cover occ11 occ12 occ13 occ21 occ22 occ23 occ31
occ32 occ33 11011.100 12 0.99 1 5 6 2 4 . 1 5 8 000110100 9 0.64 4 5 8 1 2 7
2 5 9 10.100110 9 0.21 1 2 . 1 5 8 2 3 6 110000100 8 0.54 2 5 9 5 8 11 2 5 8
111101100 15 0.37 1 3 5 6 8 9 5 7 12 11..11100 10 0.04 1 2 . . 2 3 5 8 14
100000100 17 0.58 2 3 8 5 6 7 2 . 9 100110000 9 0.38 5 8 14 1 2 8 5 8 16
1001.0100 6 0.25 4 6 8 1 . 3 1 5 6 1.110000. 17 0.34 1 . 4 3 5 9 4 5 .
111100000 3 0.23 1 2 3 4 5 6 7 8 9 000000000 15 0.87 1 2 8 2 5 6 3 7 11
1111.0010 8 0.18 1 2 4 1 . 3 2 3 . 10011011 . 7 0.72 2 4 5 2 6 7 1 2 .
110001010 14 0.49 2 5 6 4 8 9 11 12 13 101.10100 13 0.31 1 2 3 . 2 5 1 4 6
100000010 10 0.6 1 5 7 8 9 10 5 8 9 010100010 12 0.67 1 4 5 2 6 8 3 4 7
110.01110 11 0.71 1 2 3 . 4 6 1 2 7 10.11.100 10 0.26 1 2 . 1 2 . 1 5 6
110100.10 9 0.56 1 4 7 2 3 4 . 2 7 010000000 10 0.16 1 5 7 8 9 11 6 7 8
000000.00 10 0.46 1 2 5 2 5 8 . 3 4 1.0000100 12 0.69 2 . 4 5 7 9 1 2 4
100010000 11 0.42 1 2 3 4 5 6 7 8 9 000000000 12 0.42 2 5 6 5 8 9 1 3 4
0.1100110 8 0.72 1 . 5 2 5 8 1 5 7 11.100100 11 0.51 1 5 . 1 2 4 4 5 6
000000000 11 0.37 1 2 3 4 5 6 7 8 9 001100111 12 0.54 1 2 3 1 2 3 1 2 3
10.1.1100 9 0.37 1 2 . 3 . 5 1 6 8 000000000 7 0.38 1 5 7 6 8 11 1 9 14
1011.0100 8 0.35 1 5 7 2 . 5 1 3 4 100110000 9 0.86 1 2 4 2 3 6 1 2 4
11.100111 8 0.57 1 5 . 2 6 7 1 3 5
```

The data could be read into a dataframe with code as follows:

```
RDOccupancy<-read.table("RD_example.txt",
colClasses=c("character", rep("numeric",2), rep("character", 9)),
header=TRUE)
```

Note that if the file was not in the same working directory as your workspace (.RData) then you can set the working directory to the directory containing the file by using the following command before the `read.table`.

```
setwd(your working directory location here)
```

In the data file "." represents a site that was not visited on an occasion. Those "." values are read in fine because ch is read in as a character string. However, "." has also been used in the file in place of numeric values of the occ variable. Because "." is not numeric, R will coerce the input value to an NA value for each "." and will treat the column they are in as a factor. Thus, the "NA" will not be a valid numeric value for MARK, so we need to change it to a number. To avoid the coercion, the occ values were read in as characters and the following code changes all "." to "0" and then coverts the fields to numeric values:

```
 for (i in 4:12) { RDOccupancy[RDOccupancy[,i]==".",i]="0"
RDOccupancy[,i]=as.numeric(RDOccupancy[,i]) }
```

It is fine to use zero (or any numeric value) in place of missing values for session-dependent covariates as the "0's" provide no information for modeling as they are tied to un-sampled occasions. However, all values of a site-specific covariate (e.g., cover) are used, so there cannot be any missing values. Note, however that use of "0's" in the time-dependent covariates will influence predictions output by MARK for that parameter, as they will be biased low due to the zero's being included in estimating the mean for that parameter.

The code below and associated comments provide a self contained example for importing, setting up, and evaluating the any of the general robust design type models (RDOccupEG, RDOccupPE, RDOccupPG) using RMARK. Unlike standard occupancy designs, robust designs require the user to designate primary and secondary occasions using the argument `time.intervals`. For this example, we have 3 primary occasions (year) with 3 secondary sampling occasions within each year, thus, we would set our `time.intervals` as follows to represent 0 interval between secondary occasions and interval of 1 (years in this case) between primary occasions:

```
 time.intervals=c(0,0,1,0,0,1,0,0)
```

The first 0 designates the interval between the first and second sampling occasion in year 1, the second 0 designates the interval between the second and third sampling occasion in year 1, and the 1 indicated the change from primary period 1 to primary period 2. See process.data for more information on the use of `time.intervals`.

Author(s)

Bret Collier

Examples

```
data(RDOccupancy)
#
# Example of epsilon=1-gamma
test_proc=process.data(RDOccupancy,model="RDOccupEG",time.intervals=c(0,0,1,0,0,1,0,0))
test_ddl=make.design.data(test_proc)
test_ddl$Epsilon$eps=-1
test_ddl$Gamma$eps=1
p.dot=list(formula=~1)
Epsilon.random.shared=list(formula=~-1+eps, share=TRUE)
```

```
model=mark(test_proc,test_ddl,model.parameters=list(Epsilon=Epsilon.random.shared, p=p.dot))
#
# A self-contained function for evaluating a set of user-defined candidate models
run.RDExample=function()
{
# Creating list of potential predictor variables for Psi

Psi.area=list(formula=~samplearea)
Psi.cover=list(formula=~cover)
Psi.areabycover=list(formula=~samplearea*cover)
Psi.dot=list(formula=~1)
Psi.time=list(formula=~time)

# Creating list of potential predictor variables for p
# When coding formula with session-dependent (primary or secondary)
# covariates, you do NOT have to include the session identifiers (
# the ps of occps) in the model formula. You only need to specify ~occ.
# The variable suffix can be primary occasion numbers or
# primary and secondary occasion numbers.

p.dot=list(formula=~1)
p.occ=list(formula=~occ)
p.area=list(formula=~sample.area)
p.coverbyocc=list(formula=~occ*cover)

# Creating list of potential predictor variables for Gamma
# and/or Epsilon (depending on which RDOccupXX Parameterization is used)

gam.area=list(formula=~samplearea)
epsilon.area=list(formula=~samplearea)
gam.dot=list(formula=~1)
epsilon.dot=list(formula=~1)

# setting time intervals for 3 primary sessions with
# secondary session length of 3,3,3

time_intervals=c(0,0,1,0,0,1,0,0)

# Initial data processing for RMARK RDOccupPG
# (see RMARK appendix C-3 for list of RDOccupXX model paramterizations)

RD_process=process.data(RDOccupancy, model="RDOccupPG",
time.intervals=time_intervals)
RD_ddl=make.design.data(RD_process)
# Candidate model list
# 1. Occupancy, detection, and colonization are constant

model.p.dot.Psi.dot.gam.dot<-mark(RD_process, RD_ddl,
model.parameters=list(p=p.dot, Psi=Psi.dot, Gamma=gam.dot),
invisible=TRUE)

# 2. Occupancy varies by time, detection is constant,
# colonization is constant
```

```
model.p.dot.Psi.time.gam.dot<-mark(RD_process, RD_ddl,
model.parameters=list(p=p.dot, Psi=Psi.time, Gamma=gam.dot),
invisible=TRUE)

# 3. Occupancy varies by area, detection is constant,
# colonization varies by area

model.p.dot.Psi.area.gam.area<-mark(RD_process,
RD_ddl, model.parameters=list(p=p.dot, Psi=Psi.area,
Gamma=gam.area), invisible=TRUE)

# 4. Occupancy varies by cover, detection is constant,
# colonization varies by area

model.p.dot.Psi.cover.gam.area<-mark(RD_process, RD_ddl,
model.parameters=list(p=p.dot, Psi=Psi.cover, Gamma=gam.area),
invisible=TRUE)

# 5. Occupancy is constant, detection is session dependent,
# colonization is constant

model.p.occ.Psi.dot.gam.dot<-mark(RD_process, RD_ddl,
model.parameters=list(p=p.occ, Psi=Psi.dot, Gamma=gam.dot),
invisible=TRUE)

# 6. Occupancy varied by area, detection is session
# dependent, colonization is constant
model.p.occ.Psi.area.gam.dot<-mark(RD_process, RD_ddl,
model.parameters=list(p=p.occ, Psi=Psi.area, Gamma=gam.dot),
invisible=TRUE)
#
# Return model table and list of models
#
return(collect.models())
}
# This runs the 6 models above-Note that if you use
# invisible=FALSE in the above model calls
# then the mark.exe prompt screen will show as each model is run.

robustexample<-run.RDExample() #This runs the 6 models above

# Outputting model selection results
robustexample  # This will print selection results
options(width=150) # Sets page width to 100 characters
sink("results.table.txt") # Captures screen output to file

# Remove comment to see output
#print.marklist(robustexample) # Sends output to file
sink() # Returns output to screen
#
# Allows you to view results in notepad;remove # to see output
# system("notepad results.table.txt", invisible=FALSE, wait=FALSE)
```

```
# Examine the output for Model 1: Psi(.), p(.), Gamma(.)
# Opens MARK results file in text editor
#robustexample$model.p.dot.Psi.dot.gam.dot

# View beta estimates for specified model in R
robustexample$model.p.dot.Psi.dot.gam.dot$results$beta

# View real estimates for specified model in R
robustexample$model.p.dot.Psi.dot.gam.dot$results$real

# Examine the best fitting model which has a time-dependent
# effect on detection
# (Model 5: Psi(.), p(occ), Gamma(.))

# View beta estimates for specified model in R
robustexample$model.p.occ.Psi.dot.gam.dot$results$beta

# View real estimates for specified model in R
robustexample$model.p.occ.Psi.dot.gam.dot$results$real

# View estimated variance/covariance matrix in R
robustexample$model.p.occ.Psi.dot.gam.dot$results$beta.vcv


# View model averages estimates for session-dependent
# detection probabilities
model.average(robustexample, "p", vcv=TRUE)

# View model averaged estimate for Psi (Occupancy)
model.average(robustexample, "Psi", vcv=TRUE)

# View model averaged estimate for Gamma (Colonization)
model.average(robustexample, "Gamma", vcv=TRUE)

#
# Compute real estimates across the range of covariates
# for a specific model parameter using Model 6
#
# Identify indices we are interested in predicting
# see covariate.predictions for information on
# index relationship to real parameters

summary.mark(robustexample$model.p.occ.Psi.area.gam.dot, se=TRUE)
# Define data frame of covariates to be used for analysis

ha<-sort(RDOccupancy$samplearea)

# Predict parameter of interest (Psi) across the
# range of covariate data of interest

Psi.by.Area<-covariate.predictions(robustexample,
data=data.frame(samplearea=ha), indices=c(1))
```

```
# View dataframe of real parameter estimates without var-cov
# matrix printing (use str(Psi.by.Area) to evaluate structure))

Psi.by.Area[1]

#Create a simple plot using plot() and lines()

plot(Psi.by.Area$estimates$covdata, Psi.by.Area$estimates$estimate,
type="l", xlab="Patch Area", ylab="Occupancy", ylim=c(0,1))
lines(Psi.by.Area$estimates$covdata, Psi.by.Area$estimates$lcl, lty=2)
lines(Psi.by.Area$estimates$covdata, Psi.by.Area$estimates$ucl, lty=2)

# For porting graphics directly to file, see pdf() or png(),
```

RDSalamander *Robust design salamander occupancy data*

Description

A robust design occupancy data set for modelling presence/absence data for salamanders.

Format

A data frame with 40 observations (sites) on the following 2 variables.

ch a character vector containing the presence (1) and absence (0) with 2 primary occasions with 48 and 31 visits to the site

freq frequency of sites (always 1)

Details

This is a data set that I got from Gary White which is suppose to be salamander data collected with a robust design.

Examples

```
fit.RDOccupancy=function()
{
   data(RDSalamander)
   occ.p.time.eg=mark(RDSalamander,model="RDOccupEG",
      time.intervals=c(rep(0,47),1,rep(0,30)),
      model.parameters=list(p=list(formula=~session)),threads=2)
   occ.p.time.pg=mark(RDSalamander,model="RDOccupPG",
      time.intervals=c(rep(0,47),1,rep(0,30)),
      model.parameters=list(Psi=list(formula=~time),
      p=list(formula=~session)),threads=2)
```

```
   occ.p.time.pe=mark(RDSalamander,model="RDOccupPE",
      time.intervals=c(rep(0,47),1,rep(0,30)),
      model.parameters=list(Psi=list(formula=~time),
      p=list(formula=~session)),threads=2)
return(collect.models())
}
RDOcc=fit.RDOccupancy()
print(RDOcc)
```

`read.mark.binary` *Reads binary file output from MARK and returns a list of the results*

Description

Window and linux versions to read binary files created by MARK

Usage

```
read.mark.binary(filespec)
```

Arguments

`filespec`	Filename specification for binary output file from MARK;named here as markxxx.vcv

Value

List of estimates, se, lcl, ucl and var-cov matrices for beta, real and derived estimates

`beta`	Dataframe for beta parameters containing estimates, se, lcl, ucl
`beta.vcv`	variance-covariance matrix for beta estimates
`real`	Dataframe for real parameters containing estimates, se, lcl, ucl
`real.vcv`	variance-covariance matrix for real estimates
`derived`	Dataframe for derived parameters (if any) containing estimates, se, lcl, ucl
`derived.vcv`	variance-covariance matrix for derived estimates (if any)

Author(s)

Jeff Laake

See Also

`extract.mark.output`

`release.gof` *Runs RELEASE for goodness of fit test*

Description

Creates input file for RELEASE with the specified data, runs RELEASE and extracts the summary results for TEST2 and TEST3. Output file is named Releasennn.tmp where nnn is an increasing numeric value to create a unique filename.

Usage

```
release.gof(data, invisible = TRUE, title = "Release-gof", view = FALSE)
```

Arguments

`data`	processed RMark data
`invisible`	if TRUE, RELEASE run window is hidden from view
`title`	title for output
`view`	if TRUE, shows release output in a viewer window

Value

results: a dataframe giving chi-square, degrees of freedom and P value for TEST2, TEST3 and total of tests

Author(s)

Jeff Laake

Examples

```
data(dipper)
dipper.processed=process.data(dipper,groups=("sex"))
release.gof(dipper.processed)
```

remove.mark *Remove mark models from list*

Description

Remove one or more mark models from a marklist

Usage

```
remove.mark(marklist, model.numbers)
```

Arguments

marklist	an object of class "marklist" created by collect.models or merge.mark
model.numbers	vector of one more model numbers to remove from the marklist

Value

model.list: a list of mark models and a table of model results.

Author(s)

Jeff Laake

See Also

collect.models,merge.mark,run.models,model.table,dipper

Examples

```
# see example in dipper
```

rerun.mark *Runs a previous MARK model with new starting values*

Description

Runs a previous MARK model with new starting values but without specifying the model parameter formulas. This function is most useful with mark.wrapper in which a list of models is analyzed and the set of formulas are not specified for each model.

Usage

```
rerun.mark(model, data, ddl, initial, output = TRUE, title = "",
  invisible = TRUE, adjust = TRUE, se = FALSE, filename = NULL,
  prefix = "mark", default.fixed = TRUE, silent = FALSE, retry = 0,
  realvcv = FALSE, external = FALSE, threads = -1)
```

Arguments

model	previously run MARK model
data	processed dataframe used with model
ddl	design data list used with model
initial	vector of initial values for beta parameters or previously run model object of similar structure
output	If TRUE produces summary of model input and model output
title	Optional title for the MARK analysis output
invisible	if TRUE, exectution of MARK.EXE is hidden from view
adjust	if TRUE, adjusts number of parameters (npar) to number of columns in design matrix, modifies AIC and records both
se	if TRUE, se and confidence intervals are shown in summary sent to screen
filename	base filename for files created by MARK.EXE. Files are named filename.*.
prefix	base filename prefix for files created by MARK.EXE; the files are named prefixnnn.*
default.fixed	if TRUE, real parameters for which the design data have been deleted are fixed to default values
silent	if TRUE, errors that are encountered are suppressed
retry	number of reanalyses to perform with new starting values when one or more parameters are singular
realvcv	if TRUE the vcv matrix of the real parameters is extracted and stored in the model results
external	if TRUE the mark object is saved externally rather than in the workspace; the filename is kept in its place
threads	number of cpus to use with mark.exe if positive or number of cpus to remain idle if negative

Details

This is a simple function that restarts an analysis with MARK typically using another model for initial values of the beta parameters. The processed dataframe (data) and design data list (ddl) must be specified but the model.parameters are extracted from model. initial values are not optional otherwise this would be no different than the original call to mark. More complete definitions of the arguments can be found in mark or run.mark.model or make.mark.model.

Value

model: MARK model object with the base filename stored in output and the extracted results from the output file appended onto list; see mark for a detailed description of a mark object.

Author(s)

Jeff Laake

See Also

make.mark.model, run.models, extract.mark.output, adjust.parameter.count, mark, cleanup

Examples

```
## Not run:
#
# The following example will not run because the data are not included in the
# examples.  It illustrates the use of rerun.mark with mark.wrapper.  With this
# particular data set the POPAN models were having difficulty converging.  After
# running the set of models using mark.wrapper and looking at the results it
# was clear that in several instances the model did not converge.  This is easiest
# to discern by comparing nested models in the model.table.  If one model
# is nested within another,then the deviance of the model with more
# parameters should be as good or better than the smaller model.  If that
# is not the case then the model that converged can be used for initial
# values in a call to rerun.mark for the model that did not converge.
#

do.nat=function()
{
Phi.ageclass=list(formula=~ageclass)
Phi.dot=list(formula=~1)
p.area=list(formula=~area)
p.timebin.plus.area=list(formula=~timebin+area)
p.timebin.x.area=list(formula=~-1+timebin:area)
pent.ageclass=list(formula=~ageclass)
pent.ageclass.plus.EN=list(formula=~ageclass+EN)
pent.ageclass.plus.diffEN=list(formula=~ageclass+EN92+EN97+EN02)
cml=create.model.list("POPAN")
nat=mark.wrapper(cml,data=zc.proc,ddl=zc.ddl,
  invisible=FALSE,initial=1,retry=2)
return(nat)
}
nat=do.nat()
# model list
#            Phi                   p                      pent
#1  Phi.ageclass              p.area             pent.ageclass
#2  Phi.ageclass              p.area pent.ageclass.plus.diffEN
#3  Phi.ageclass              p.area     pent.ageclass.plus.EN
#4  Phi.ageclass p.timebin.plus.area             pent.ageclass
#5  Phi.ageclass p.timebin.plus.area pent.ageclass.plus.diffEN
#6  Phi.ageclass p.timebin.plus.area     pent.ageclass.plus.EN
#7  Phi.ageclass    p.timebin.x.area             pent.ageclass
#8  Phi.ageclass    p.timebin.x.area pent.ageclass.plus.diffEN
#9  Phi.ageclass    p.timebin.x.area     pent.ageclass.plus.EN
#10      Phi.dot              p.area             pent.ageclass
#11      Phi.dot              p.area pent.ageclass.plus.diffEN
#12      Phi.dot              p.area     pent.ageclass.plus.EN
#13      Phi.dot p.timebin.plus.area             pent.ageclass
#14      Phi.dot p.timebin.plus.area pent.ageclass.plus.diffEN
#15      Phi.dot p.timebin.plus.area     pent.ageclass.plus.EN
```

```
#16      Phi.dot    p.timebin.x.area             pent.ageclass
#17      Phi.dot    p.timebin.x.area pent.ageclass.plus.diffEN
#18      Phi.dot    p.timebin.x.area     pent.ageclass.plus.EN
#
# use model 9 as starting values for model 7
nat[[7]]= rerun.mark(nat[[7]],data=zc.proc,ddl=zc.ddl,initial=nat[[9]])
# use model 3 as starting values for model 1
nat[[1]]= rerun.mark(nat[[1]],data=zc.proc,ddl=zc.ddl,initial=nat[[3]])
# use model 14 as starting values for model 15
nat[[15]]= rerun.mark(nat[[15]],data=zc.proc,ddl=zc.ddl,initial=nat[[14]])
# use model 5 as starting values for model 6
nat[[6]]= rerun.mark(nat[[6]],data=zc.proc,ddl=zc.ddl,initial=nat[[5]])
# use model 10 as starting values for model 11
nat[[11]]= rerun.mark(nat[[11]],data=zc.proc,ddl=zc.ddl,initial=nat[[10]])
# use model 10 as starting values for model 12
nat[[12]]= rerun.mark(nat[[12]],data=zc.proc,ddl=zc.ddl,initial=nat[[10]])
# reconstruct model table with new results
nat$model.table=model.table(nat[1:18])
# show new model table
nat

## End(Not run)
```

`robust` *Robust design example data*

Description

A robust design example data set that accompanies MARK as an example analysis using the various models for the robust design.

Format

A data frame with 668 observations on the following 2 variables.

ch a character vector containing the encounter history

freq the number of critters with that capture history

Details

This is a data set that accompanies program MARK as an example for robust models. The data are entered with the summary format using the variable `freq` which represents the number of critters with that capture (encounter) history. The data set represents a robust design with 5 primary occasions and within each primary occasion the number of secondary occasions is 2,2,4,5,2 respectively. This is represented with the `time.intervals` argument of `process.data` which are 0,1,0,1,0,0,0,1,0,0,0,0,1,0. The 0 time intervals represent the secondary sessions in which the population is assumed to be closed. The non-zero values are the time intervals between the primary occasions. They are all 1 in this example but they can have different non-zero values. The code determines the structure of the robust design based on the time intervals. The intervals must begin

and end with at least one 0 and there must be at least one 0 between any 2 non-zero elements. The number of occasions in a secondary session is one plus the number of contiguous zeros.

Examples

```
data(robust)
run.robust=function()
{
#
# data from Robust.dbf with MARK
# 5 primary sessions with secondary sessions of length 2,2,4,5,2
#
time.intervals=c(0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0)
#
# Random emigration, p=c varies by time and session, S by time
#
S.time=list(formula=~time)
p.time.session=list(formula=~-1+session:time,share=TRUE)
GammaDoublePrime.random=list(formula=~time,share=TRUE)
model.1=mark(data = robust, model = "Robust",
            time.intervals=time.intervals,
            model.parameters=list(S=S.time,
            GammaDoublePrime=GammaDoublePrime.random,p=p.time.session),threads=2)
#
# Random emigration, p varies by session, uses Mh but pi fixed to 1,
# S by time.This model is in the example Robust with MARK but it is
# a silly example because it uses the heterogeneity model but then fixes
# pi=1 which means there is no heterogeneity.Probably the data were
# not generated under Mh.  See results of model.2.b
#
pi.fixed=list(formula=~1,fixed=1)
p.session=list(formula=~-1+session,share=TRUE)
model.2.a=mark(data = robust, model = "RDHet",
            time.intervals=time.intervals,
            model.parameters=list(S=S.time,
            GammaDoublePrime=GammaDoublePrime.random,
            p=p.session,pi=pi.fixed),threads=2)
#
# Random emigration, p varies by session, uses Mh and in this
# case pi varies and so does p across
# mixtures with an additive session effect.
#
pi.dot=list(formula=~1)
p.session.mixture=list(formula=~session+mixture,share=TRUE)
model.2.b=mark(data = robust, model = "RDHet",
            time.intervals=time.intervals,
            model.parameters=list(S=S.time,
            GammaDoublePrime=GammaDoublePrime.random,
            p=p.session.mixture,pi=pi.dot),threads=2)
#
# Markov constant emigration rates, pi varies by session,
# p=c varies by session, S constant
```

```
# This model is in the example Robust with MARK
# but it is a silly example because it
# uses the heterogeneity model but then fixes pi=1
# which means there is no heterogeneity.
# Probably the data were not generated under Mh.
# See results of model.3.b
#
S.dot=list(formula=~1)
pi.session=list(formula=~session)
p.session=list(formula=~-1+session,share=TRUE)
GammaDoublePrime.dot=list(formula=~1)
GammaPrime.dot=list(formula=~1)
model.3.a=mark(data = robust, model = "RDHet",
            time.intervals=time.intervals,
            model.parameters=list(S=S.dot,
            GammaPrime=GammaPrime.dot,
            GammaDoublePrime=GammaDoublePrime.dot,
            p=p.session,pi=pi.session),threads=2)
#
# Markov constant emigration rates, pi varies by session,
# p=c varies by session+mixture, S constant. This is model.3.a
# but allows pi into the model by varying p/c by mixture.
#
S.dot=list(formula=~1)
pi.session=list(formula=~session)
GammaDoublePrime.dot=list(formula=~1)
GammaPrime.dot=list(formula=~1)
model.3.b=mark(data = robust, model = "RDHet",
            time.intervals=time.intervals,
            model.parameters=list(S=S.dot,
            GammaPrime=GammaPrime.dot,
            GammaDoublePrime=GammaDoublePrime.dot,
            p=p.session.mixture,pi=pi.session),threads=2)
#
# Huggins Random emigration, p=c varies by time and session,
# S by time
# Beware that this model is not quite the same
# as the others above that say random emigration because
# the rates have been fixed for the last 2 occasions.
# That was done with PIMS in the MARK example and
# here it is done by binning the times so that times 3 and 4
# are in the same bin, so the time model
# has 3 levels (1,2, and 3-4).  By doing so the parameters
# become identifiable but this may not be
# reasonable depending on the particulars of the data.
# Note that the same time binning must be done both for
# GammaPrime and GammaDoublePrime because the parameters are
# the same in the random emigration model.  If you
# forget to bin one of the parameters across time it will fit
# a model but it won't be what you expect as it will
# not share parameters.  Note the use of the argument "right".
# This controls whether binning is inclusive on the right (right=TRUE)
# or on the left (right=FALSE).  Using "right" nested in the list
```

```
# of design parameters is equivalent to using it as a calling
# argument to make.design.data or add.design.data.
#
S.time=list(formula=~time)
p.time.session=list(formula=~-1+session:time,share=TRUE)
GammaDoublePrime.random=list(formula=~time,share=TRUE)
model.4=mark(data = robust, model = "RDHuggins",
        time.intervals=time.intervals,design.parameters=
        list(GammaDoublePrime=list(time.bins=c(1,2,5))),
        right=FALSE, model.parameters=
        list(S=S.time,GammaDoublePrime=GammaDoublePrime.random,
        p=p.time.session),threads=2)

return(collect.models())
}
robust.results=run.robust()
#
#  You will receive a warning message that the model list
#  includes models of different types which are not compatible
#  for comparisons of AIC.  That is because
#  the runs include closed models which include N
#  in the likelihood and Huggins models which don't include
#  N in the likelihood.  That can be avoided by running
#  the two types of models in different sets.
#
robust.results
```

`run.mark.model` *Runs analysis with MARK model using MARK.EXE*

Description

Passes input file from model (`model$input`) to MARK, runs MARK, gets `output` and extracts relevant values into `results` which is appended to the `mark` model object.

Usage

```
run.mark.model(model, invisible = FALSE, adjust = TRUE, filename = NULL,
  prefix = "mark", realvcv = FALSE, delete = FALSE, external = FALSE,
  threads = -1, ignore.stderr = FALSE)
```

Arguments

`model`	MARK model created by `make.mark.model`
`invisible`	if TRUE, exectution of MARK.EXE is hidden from view
`adjust`	if TRUE, adjusts number of parameters (npar) to number of columns in design matrix, modifies AIC and records both
`filename`	base filename for files created by MARK.EXE. Files are named filename.*.

prefix	base filename prefix for files created by MARK.EXE; the files are named prefixnnn.*
realvcv	if TRUE the vcv matrix of the real parameters is extracted and stored in the model results
delete	if TRUE the output files are deleted after the results are extracted
external	if TRUE the mark object is saved externally rather than in the workspace; the filename is kept in its place
threads	number of cpus to use with mark.exe if positive or number of cpus to remain idle if negative
ignore.stderr	If set TRUE, messages from mark.exe are suppressed; they are automatically suppressed with Rterm

Details

This is a rather simple function that initiates the analysis with MARK and extracts the output. An analysis was split into two functions make.mark.model and run.mark.model to allow a set of models to be created and then run individually or collectively with run.models. By default, the execution of MARK.EXE will appear in a separate window in which the progress can be monitored. The window can be suppressed by setting the argument invisible=TRUE. The function returns a mark object and it should be assigned to the same object to replace the original model (e.g., mymodel=run.mark.model(mymodel)). The element output is the base filename that links the objects to the output files stored in the same directory as the R workspace. To removed unneeded output files after deleting mark objects in the workspace, see cleanup. results is a list of specific output values that are extracted from the output. In extracting the results, the number of parameters can be adjusted (adjust=TRUE) to match the number of columns in the design matrix, which assumes that it is full rank and that all of the parameters are estimable and not confounded. This can be useful if that assumption is true, because on occasion MARK.EXE will report an incorrect number of parameters in some cases in which the parameters are at boundaries (e.g., 0 or 1 for probabilities). If the true parameter count is neither that reported by MARK.EXE nor the number of columns in the design matrix, then it can be adjusted using adjust.parameter.count.

If filename is assigned a value it is used to specify files with those names. This is most useful to capture output from a model that has already been run. If it finds the files with those names already exists, it will ask if the results should be extracted from the files rather than re-running the models.

Value

model: MARK model object with the base filename stored in output and the extracted results from the output file appended onto list; see mark for a detailed description of a mark object.

Author(s)

Jeff Laake

See Also

make.mark.model, run.models, extract.mark.output, adjust.parameter.count, mark, cleanup

Examples

```
test=function()
{
  data(dipper)
  for(sex in unique(dipper$sex))
  {
  x=dipper[dipper$sex==sex,]
  x.proc=process.data(x,model="CJS")
  x.ddl=make.design.data(x.proc)
  Phi.dot=list(formula=~1)
  Phi.time=list(formula=~time)
  p.dot=list(formula=~1)
  p.time=list(formula=~time)
  cml=create.model.list("CJS")
  x.results=mark.wrapper(cml,data=x.proc,ddl=x.ddl,prefix=sex)
  assign(paste(sex,"results",sep="."),x.results)
  }
  rm(Male.results,Female.results,x.results)
}
test()
cleanup(ask=FALSE,prefix="Male")
cleanup(ask=FALSE,prefix="Female")
```

---

run.models *Runs a set of MARK models*

Description

Runs either a collection of models as defined in `model.list` or runs all defined MARK object models in the frame of the calling function with no `output` (`model.list=NULL`) or just those of a particular type (e.g., `type="CJS"`)

Usage

```
run.models(model.list = NULL, type = NULL, save = TRUE, ...)
```

Arguments

`model.list`	either a vector of model names or NULL to run all MARK models possibly of a particular `type`
`type`	either a model type (eg "CJS", "Burnham" or "Barker") or NULL for all types
`save`	if TRUE, the R data directory is saved (i.e., `save.image()`) between analyses to enable interruption without losing analyses that have already been run
`...`	any additional parameters to be passed to `run.mark.model`

Details

The model names in `model.list` must be in the frame of the function that calls `run.models`. If `model.list=NULL` or the MARK models are collected from the frame of the calling function (the parent). If `type` is specified only the models of that type (e.g., "CJS") are run. In each case the models are run and saved in the parent frame.

Value

None; models are stored in parent frame.

Author(s)

Jeff Laake

See Also

`collect.model.names`, `run.mark.model`

`salamander` *Salamander occupancy data*

Description

An occupancy data set for modelling presence/absence data for salamanders.

Format

A data frame with 39 observations (sites) on the following 2 variables.

ch a character vector containing the presence (1) and absence (0) for each visit to the site

freq frequency of sites (always 1)

Details

This is a data set that accompanies program PRESENCE and is explained on page 99 of MacKenzie et al. (2006).

References

MacKenzie, D.I., Nichols, J. D., Royle, J.A., Pollock, K.H., Bailey, L.L., and Hines, J.E. 2006. Occupancy Estimation and Modeling: Inferring Patterns and Dynamics of Species Occurence. Elsevier, Inc. 324p.

Examples

```
do.salamander=function()
{
   data(salamander)
   occ.p.dot=mark(salamander,model="Occupancy")
   occ.p.time=mark(salamander,model="Occupancy",
         model.parameters=list(p=list(formula=~time)))
   occ.p.mixture=mark(salamander,model="OccupHet",
         model.parameters=list(p=list(formula=~mixture)))
   return(collect.models())
}
salamander.results=do.salamander()
print(salamander.results)
```

---

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) defined in the call to make.mark.model

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

Details

The primary difference in setting up models for MARK is the number and types of parameters that are included in the model. This function sets up the list of parameters used in the model and defines values for each parameter that affect how the PIM and design data are structured in the input file for program MARK. Some of the values of the parameter list are user specified such as `formula`, `link`,`fixed` so this function only adds to the list of values that are not specified by the user. That is, it takes the input argument `parameters` and adds list elements for parameters not specified by the user and adds default values for each type of parameter and then returns the modified list. The structure of the argument `parameters` and the return value of this function are the same as the structure of the argument `parameters` in `make.mark.model` and argument `model.parameters` in `mark`. They are lists with an element for each type of parameter in the model and the name of each list element is the parameter name (e.g., "p", "Phi","S", etc). For each parameter there are a list of values (e.g., formula, link, num etc as defined below). Thus `parameters` is a list of lists.

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

---

splitCH *Split/collapse capture histories*

Description

splitCH will split a character string vector of capture histories into a matrix. 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")
```

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

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)
```

---

store *Store models externally or restore to workspace from external storage*

Description

Stores/restores all mark model objects in a marklist either to or from external storage.

Usage

```
store(x)
```

Arguments

x	marklist of models

Details

For `store`, each mark model is stored externally and the object in the list is replaced with the filename of the object. `restore` does the opposite of storing the saved external object into the marklist and then deleting the saved file.

Value

A modified marklist to replace the previous marklist specified as the argument.

Author(s)

Jeff Laake

strip.comments *Strip comments*

Description

Read in file and strip out comments and blank lines.

Usage

```
strip.comments(inp.filename, use.comments = TRUE, header = TRUE)
```

Arguments

inp.filename	name of input file; inp extension is assumed and does not need to be specified
use.comments	if TRUE values within /* and */ on data lines are used as row.names for the RMark dataframe. Only use this option if they are unique values.
header	if TRUE, input file has header line with field names

Value

rn	row names
out.filename	output filename

Author(s)

Jeff Laake

summary.mark *Summary of MARK model parameters and results*

Description

Creates a summary object of either a MARK model input or model output which includes number of parameters, deviance, AICc, the beta and real parameter estimates and optionally standard errors, confidence intervals and variance-covariance matrices. If there are several groups in the data, the output is structured by group.

Usage

```
## S3 method for class 'mark'
summary(object,...,se=FALSE,vc=FALSE,showall=TRUE,show.fixed=FALSE,brief=FALSE)
       ## S3 method for class 'mark'
coef(object,...)
       ## S3 method for class 'summary.mark'
print(x,...)
```

Arguments

object	a MARK model object
...	additional non-specified argument for S3 generic function
se	if FALSE the real parameter estimates are output in PIM format (eg. triangular format); if TRUE, they are displayed as a list with se and confidence interval
vc	if TRUE the v-c matrix of the betas is included
showall	if FALSE it only returns the values of each unique parameter value
show.fixed	if FALSE, each fixed value given NA; otherwise the fixed real value is used. If se=TRUE, default for show.fixed=TRUE
brief	if TRUE, does not show real parameter estimates
x	list resulting from call to summary

Details

The structure of the summary of the real parameters depends on the type of model and the value of the argument se and showall. If se=F then only the estimates of the real parameters are shown and they are summarized the result element reals in PIM format. The structure of reals depends on whether the PIMS are upper triangular ("Triang") or a row ("Square" although not really square). For the upper triangular format, the values are passed back as a list of matrices where the list is a list of parameter types (eg Phi and p) and within each type is a list for each group containing the pim as an upper triangular matrix containing the real parameter estimate. For square matrices, reals is a list of matrices with a list element for each parameter type, but there is not a second list layer for groups because in the returned matrix each group is a row in the matrix of real estimates. If se=TRUE then estimates, standard error (se), lower and upper confidence limits (lcl, ucl) and a "Fixed" indicator is passed for each real parameter. If the pims for the model were simplified to represent the unique real parameters (unique rows in the design matrix), then it is possible to restict the summary to only the unique parameters with showall=FALSE. This argument only has an affect if se=TRUE. If showall=FALSE, reals is returned as a dataframe of the unique real parameters specified in the model. This does not mean they will all have unique values and it includes all "Fixed" real parameters and any real parameters that cannot be simplified in the case of parameters such as "pent" in POPAN or "Psi" in "Multistrata" that use the multinomial logit link. Use of showall=FALSE is of limited use but provided for completeness. In most cases the default of showall=TRUE will be satisfactory. In this case, reals is a list of dataframes with a list element for each parameter type. The dataframe contains the estimate, se,lcl, ucl,fixed and the associated default design data for that parameter (eg time,age, cohort etc). The advantage of retrieving the reals in this format is that it is the same regardless of the model, so it enables model averaging the real parameters over different models with differing numbers of unique real parameters.

Value

A list with each of the summarized objects that depends on the argument values. Only the first 4 are given if it is a summary of a model that has not been run.

model	type of model (e.g., CJS)
title	user define title if any
model.name	descriptive name of fitted model

call	call to make.mark.model used to construct the model
npar	number of fitted parameters
lnl	-2xLog Likelihood value
npar	Number of parameters (always the number of columns in design matrix)
chat	Value of over-dispersion constant if not equal to 1
npar.unadjusted	number of estimated parameters from MARK if different than npar
AICc	Small sample corrected AIC using npar; named qAICc if chat not equal to 1
AICc.unadjusted	Small sample corrected AIC using npar.unadjusted; prefix of q if chat not equal to 1
beta	dataframe of beta parameters with estimate, se, lcl, ucl
vcv	variance-covariance matrix for beta
reals	list of lists, dataframes or matrices depending on value of se and the type of model (triangular versus square PIMS) (see details above)

Author(s)

Jeff Laake

---

summary_ch *Provides a summary for the capture histories*

Description

For each release (initial capture) cohort, the number of recaptured (resighted) individuals from that cohort is tallied for each of the following occasions. A summary table with number released (initially caught) and the number recaptured is given for each group if bygroup=TRUE.

Usage

```
summary_ch(x, bygroup = TRUE, marray = FALSE)
```

Arguments

x	Processed data list; resulting value from process.data
bygroup	if TRUE, summary tables are created for each group defined in the data
marray	if TRUE, summary tables are m-arrays as in MARK

Value

list of dataframes (one for each group in the data); each dataframe has rows for each release cohort and columns for each recapture occasion. The rows and columns are labelled with the occasion time labels. If `marray==FALSE` the first column is the number initially released and the remaining columns (one for each recapture/resighting occasion) are the number recaught in each of the following occasions and the number caught in at least one of the occasions. If `marray==TRUE` the first column is the number released which includes those initially released and ones released after recapture from a previous cohort. The remaining columns are the number first recaught in each of the following occasions. Once re-caught they become one of the following rows (ie release-recap pairs) unless it is the last time they were captured and they were not released (eg negative frequency).

Author(s)

Jeff Laake

Examples

```
data(dipper)
dipper.processed=process.data(dipper,groups=("sex"))
summary_ch(dipper.processed)
#$sexFemale
#  Released 2  3 4  5  6  7 Total
#1       10 5  3 3  2  1  0     6
#2       29 0 11 6  6  4  2    11
#3       27 0  0 9  5  3  2     9
#4       23 0  0 0 11  7  4    13
#5       19 0  0 0  0 12  6    12
#6       23 0  0 0  0  0 11    11
#
#$sexMale
#  Released 2 3  4  5  6  7 Total
#1       12 6 3  2  1  1  0     7
#2       20 0 9  2  1  0  0     9
#3       25 0 0 13  6  2  0    14
#4       22 0 0  0 15  9  7    16
#5       22 0 0  0  0 13 10    13
#6       23 0 0  0  0  0 12    12
summary_ch(dipper.processed,marray=TRUE)
#$sexFemale
#  Released 2  3  4  5  6  7 Total
#1       10 5  1  0  0  0  0     6
#2       34 0 13  1  0  0  0    14
#3       41 0  0 17  1  0  0    18
#4       41 0  0  0 23  1  1    25
#5       43 0  0  0  0 26  0    26
#6       50 0  0  0  0  0 24    24
#
#$sexMale
#  Released 2  3  4  5  6  7 Total
#1       12 6  1  0  0  0  0     7
#2       26 0 11  0  0  0  0    11
```

```
#3       37 0  0 17  1  0  0    18
#4       39 0  0  0 22  0  1    23
#5       45 0  0  0  0 25  0    25
#6       48 0  0  0  0  0 28    28
```

`TransitionMatrix` *Multi-state Transition Functions*

Description

TransitionMatrix: Creates a transition matrix of movement parameters for a multi-state(strata) model. It computes all Psi values for a multi-strata mark model and constructs a transition matrix. Standard errors and confidence intervals can also be obtained.

Usage

```
TransitionMatrix(x,vcv.real=NULL)

        find.possible.transitions(ch)

        transition.pairs(ch)
```

Arguments

`x`	Estimate table from `get.real` with a single record for each possible transition
`vcv.real`	optional variance-covariance matrix from the call to `get.real`
`ch`	vector of capture history strings for a multi-state analysis

Details

find.possible.transitions: Finds possible transitions; essentially it identifies where stratum label A and B are in the same ch for all labels but the the transition could be from A to B or B to A or even ACB which is really an A to C and C to B transition.

transition.pairs: Computes counts of transition pairs. The rows are the "from stratum" and the columns are the "to stratum". So AB would be in the first row second column and BA would be in the second row first column. All intervening 0s are ignored. These are transition pairs so AB0C is A to B and B to C but not A to C.

Value

TransitionMatrix: returns either a transition matrix (vcv.real=NULL) or a list of matrices (vcv.real specified) named TransitionMat (transition matrix), se.TransitionMat (se of each transition), lcl.TransitionMat (lower confidence interval limit for each transition), and ucl.TransitionMat (upper confidence interval limit for each transition). find.possible.transitions returns a 0/1 table where 1 means that t both values are in one or more ch strings and transition.pairs returns a table of counts of transition pairs.

Author(s)

Jeff Laake

See Also

get.real

Examples

```
data(mstrata)
# Show possible transitions in first 15 ch values
find.possible.transitions(mstrata$ch[1:15])
# Show transtion pairs for same data
transition.pairs(mstrata$ch[1:15])
#limit transtions to 2 and 3 character values for first 30 ch
transition.pairs(substr(mstrata$ch[1:30],2,3))

# fit the sequence of multistrata models as shown for ?mstrata
run.mstrata=function()
{
#
# Process data
#
mstrata.processed=process.data(mstrata,model="Multistrata")
#
# Create default design data
#
mstrata.ddl=make.design.data(mstrata.processed)
#
#  Define range of models for S; note that the betas will differ from the output
#  in MARK for the ~stratum = S(s) because the design matrix is defined using
#  treatment contrasts for factors so the intercept is stratum A and the other
#  two estimates represent the amount that survival for B abd C differ from A.
#  You can use force the approach used in MARK with the formula ~-1+stratum which
#  creates 3 separate Betas - one for A,B and C.
#
S.stratum=list(formula=~stratum)
S.stratumxtime=list(formula=~stratum*time)
#
#  Define range of models for p
#
p.stratum=list(formula=~stratum)
#
#  Define range of models for Psi; what is denoted as s for Psi
#  in the Mark example for Psi is accomplished by -1+stratum:tostratum which
#  nests tostratum within stratum.  Likewise, to get s*t as noted in MARK you
#  want ~-1+stratum:tostratum:time with time nested in tostratum nested in
#  stratum.
#
Psi.s=list(formula=~-1+stratum:tostratum)
Psi.sxtime=list(formula=~-1+stratum:tostratum:time)
```

```
#
# Create model list and run assortment of models
#
model.list=create.model.list("Multistrata")
#
# Add on specific models that are paired with fixed p's to remove confounding
#
p.stratumxtime=list(formula=~stratum*time)
p.stratumxtime.fixed=list(formula=~stratum*time,fixed=list(time=4,value=1))
model.list=rbind(model.list,c(S="S.stratumxtime",p="p.stratumxtime.fixed",
               Psi="Psi.sxtime"))
model.list=rbind(model.list,c(S="S.stratum",p="p.stratumxtime",Psi="Psi.s"))
#
# Run the list of models
#
mstrata.results=mark.wrapper(model.list,data=mstrata.processed,ddl=mstrata.ddl)
#
# Return model table and list of models
#
return(mstrata.results)
}
mstrata.results=run.mstrata()
mstrata.results
# for the best model, get.real to get a list containing all Psi estimates
#  and the v-c matrix
Psilist=get.real(mstrata.results[[1]],"Psi",vcv=TRUE)
Psivalues=Psilist$estimates
# call Transition matrix using values from time==1; the call to the function
# must only contain one record for each possible transition. An error message is
# given if not the case
TransitionMatrix(Psivalues[Psivalues$time==1,])
# call it again but specify the vc matrix to get se and conf interval
TransitionMatrix(Psivalues[Psivalues$time==1,],vcv.real=Psilist$vcv.real)
```

`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

---

var.components *Variance components estimation*

Description

Computes estimated effects, standard errors and process variance for a set of estimates

Usage

```
var.components(theta, design, vcv, alpha = 0.05, upper = 10 * max(vcv),
  LAPACK = TRUE)
```

Arguments

theta	vector of parameter estimates
design	design matrix for combining parameter estimates
vcv	estimated variance-covariance matrix for parameters
alpha	sets 1-alpha confidence limit on sigma
upper	upper limit for process variance
LAPACK	argument passed to call to qr for qr decomposition and inversion

Details

Computes estimated effects, standard errors and process variance for a set of estimates using the method of moments estimator described by Burnham and White (2002). The design matrix specifies the manner in which the estimates (theta) are combined. The number of rows of the design matrix must match the length of theta.

If you select specific values of theta, you must select the equivalent sub-matrix of the variance-covariance matrix. For instance, if the parameter indices are $estimates[c(1:5,8)] then the appropriate definition of the vcv matrix would be vcv=vcv[c(1:5,8), c(1:5,8)], if vcv is nxn for n estimates. Note that get.real will only return the vcv matrix of the unique reals so the dimensions of estimates and vcv will not always match as in the example below where estimates has 21 rows but with the time model there are only 6 unique Phis so vcv is 6x6.

To get a mean estimate use a column matrix of 1's (e.g., `design=matrix(1,ncol=1,nrow=length(theta))`. The function returns a list with the estimates of the coefficients for the design matrix (`beta`) with one value per column in the design matrix and the variance-covariance matrix (`vcv.beta`) for the `beta` estimates. The process variance is returned as `sigma`.

Value

A list with the following elements

`sigmasq`	process variance estimate and confidence interval; estimate may be <0
`sigma`	sqrt of process variance; set to o if sigmasq<0
`beta`	dataframe with estimates and standard errors of betas for design
`betarand`	dataframe of shrinkage estimates
`vcv.beta`	variance-covariance matrix for beta
`GTrace`	trace of matrix G

Author(s)

Jeff Laake; Ben Augustine

References

BURNHAM, K. P. and G. C. WHITE. 2002. Evaluation of some random effects methodology applicable to bird ringing data. Journal of Applied Statistics 29: 245-264.

Examples

```
data(dipper)
md=mark(dipper,model.parameters=list(Phi=list(formula=~time)))
md$results$AICc
zz=get.real(md,"Phi",vcv=TRUE)
z=zz$estimates$estimate[1:6]
vcv=zz$vcv.real
varc=var.components(z,design=matrix(rep(1,length(z)),ncol=1),vcv)
df=md$design.data$Phi
shrinkest=data.frame(time=1:6,value=varc$betarand$estimate)
df=merge(df,shrinkest,by="time")
md=mark(dipper,model.parameters=list(Phi=list(formula=~time,
  fixed=list(index=df$par.index,value=df$value))),adjust=FALSE)
npar=md$results$npar+varc$GTrace
md$results$lnl+2*(npar + (npar*(npar+1))/(md$results$n-npar-1))
```

`var.components.reml` *Variance components estimation using REML or maximum likelihood*

Description

Computes estimated effects, standard errors and variance components for a set of estimates

Usage

```
var.components.reml(theta, design, vcv = NULL, rdesign = NULL,
  initial = NULL, interval = c(-25, 10), REML = TRUE)
```

Arguments

`theta`	vector of parameter estimates
`design`	design matrix for fixed effects combining parameter estimates
`vcv`	estimated variance-covariance matrix for parameters
`rdesign`	design matrix for random effect (do not use intercept form; eg use ~-1+year instead of ~year); if NULL fits only iid error
`initial`	initial values for variance components
`interval`	interval bounds for log(sigma) to help optimization from going awry
`REML`	if TRUE uses reml else maximum likelihood

Details

The function `var.components` uses method of moments to estimate a single process variance but cannot fit a more complex example. It can only estimate an iid process variance. However, if you have a more complicated structure in which you have random year effects and want to estimate a fixed age effect then `var.components` will not work because it will assume an iid error rather than allowing a common error for each year as well as an iid error. This function uses restricted maximum likelihood (reml) or maximum likelihood to fit a fixed effects model with an optional random effects structure. The example below provides an illustration as to how this can be useful.

Value

A list with the following elements

`neglnl`	negative log-likelihood for fitted model
`AICc`	small sample corrected AIC for model selection
`sigma`	variance component estimates; if rdesign=NULL, only an iid error; otherwise, iid error and random effect error
`beta`	dataframe with estimates and standard errors of betas for design
`vcv.beta`	variance-covariance matrix for beta

Author(s)

Jeff Laake

Examples

```
# Use dipper data with an age (0,1+)/time model for Phi
data(dipper)
dipper.proc=process.data(dipper,model="CJS")
dipper.ddl=make.design.data(dipper.proc,
   parameters=list(Phi=list(age.bins=c(0,.5,6))))
levels(dipper.ddl$Phi$age)=c("age0","age1+")
md=mark(dipper,model.parameters=list(Phi=list(formula=~time+age)))
# extract the estimates of Phi
zz=get.real(md,"Phi",vcv=TRUE)
# assign age to use same intervals as these are not copied
# across into the dataframe from get.real
zz$estimates$age=cut(zz$estimates$Age,c(0,.5,6),include=TRUE)
levels(zz$estimates$age)=c("age0","age1+")
z=zz$estimates
# Fit age fixed effects with random year component and an iid error
var.components.reml(z$estimate,design=model.matrix(~-1+age,z),
        zz$vcv,rdesign=model.matrix(~-1+time,z))
# Fitted model assuming no covariance structure to compare to
# results with lme
xx=var.components.reml(z$estimate,design=model.matrix(~-1+age,z),
 matrix(0,nrow=nrow(zz$vcv),ncol=ncol(zz$vcv)),
 rdesign=model.matrix(~-1+time,z))
xx
sqrt(xx$sigmasq)
library(nlme)
lme(estimate~-1+age,data=z,random=~1|time)
```

---

weta *Occupancy data for Mahoenui Giant Weta*

Description

An occupancy data set for modelling presence/absence data for salamanders.

Format

A data frame with 72 observations (sites) on the following 7 variables.

ch a character vector containing the presence (1) and absence (0), or (.) not visited for each of 5 visits to the site

Browse 0/1 dummy variable to indicate browsing

Obs1 observer number for visit 1; . used when site not visited

Obs2 observer number for visit 2; . used when site not visited

Obs3 observer number for visit 3; . used when site not visited

Obs4 observer number for visit 4; . used when site not visited

Obs5 observer number for visit 5; . used when site not visited

Details

This is a data set that accompanies program PRESENCE and is explained on pages 116-122 of MacKenzie et al. (2006).

References

MacKenzie, D.I., Nichols, J. D., Royle, J.A., Pollock, K.H., Bailey, L.L., and Hines, J.E. 2006. Occupancy Estimation and Modeling: Inferring Patterns and Dynamics of Species Occurence. Elsevier, Inc. 324p.

Examples

```
#  The data can be imported with the following command using the
#  tab-delimited weta.txt file in the data subdirectory.
#   weta=import.chdata("weta.txt",field.types=c(rep("f",6)))
#  Below is the first few lines of the data file that was constructed
#  from the .xls file that accompanies PRESENCE.
#ch Browse Obs1 Obs2 Obs3 Obs4 Obs5
#0000. 1 1 3 2 3 .
#0000. 1 1 3 2 3 .
#0001. 1 1 3 2 3 .
#0000. 0 1 3 2 3 .
#0000. 1 1 3 2 3 .
#0000. 0 1 3 2 3 .
#

# retrieve weta data
data(weta)
# Create function to fit the 18 models in the book
fit.weta.models=function()
{
#  use make.time.factor to create time-varying dummy variables Obs1 and Obs2
#  observer 3 is used as the intercept
   weta=make.time.factor(weta,"Obs",1:5,intercept=3)
#  Process data and use Browse covariate to group sites; it could have also
#  been used an individual covariate because it is a 0/1 variable.
   weta.process=process.data(weta,model="Occupancy",groups="Browse")
   weta.ddl=make.design.data(weta.process)
#  time factor variable copied to Day to match names used in book
   weta.ddl$p$Day=weta.ddl$p$time
# Define p models
   p.dot=list(formula=~1)
   p.day=list(formula=~Day)
   p.obs=list(formula=~Obs1+Obs2)
   p.browse=list(formula=~Browse)
```

```
   p.day.obs=list(formula=~Day+Obs1+Obs2)
   p.day.browse=list(formula=~Day+Browse)
   p.obs.browse=list(formula=~Obs1+Obs2+Browse)
   p.day.obs.browse=list(formula=~Day+Obs1+Obs2+Browse)
# Define Psi models
   Psi.dot=list(formula=~1)
   Psi.browse=list(formula=~Browse)
# Create model list
   cml=create.model.list("Occupancy")
# Run and return marklist of models
   return(mark.wrapper(cml,data=weta.process,ddl=weta.ddl))
}
weta.models=fit.weta.models()
# Modify the model table to show -2lnl and use AIC rather than AICc
weta.models$model.table=model.table(weta.models,use.AIC=TRUE,use.lnl=TRUE)
# Show new model table which duplicates the results except they have
# some type of error with the model Psi(.)P(Obs+Browse) which should have
# 5 parameters rather than 4 and the -2lnl also doesn't agree with the results here
weta.models
#
# display beta vcv matrix of the Psi parameters (intercept + browse=1)
# matches what is shown on pg 122 of Occupancy book
weta.models[[7]]$result$beta.vcv[8:9,8:9]
# compute variance-covariance matrix of Psi0(6; unbrowsed) ,Psi1(7; browsed)
vcv.psi=get.real(weta.models[[7]],"Psi",vcv=TRUE)$vcv.real
vcv.psi
# Compute proportion unbrowsed and browsed
prop.browse=c(37,35)/72
prop.browse
# compute std error of overall estimate as shown on pg 121-122
sqrt(sum(prop.browse^2*diag(vcv.psi)))
# compute std error and correctly include covariance between Psi0 and Psi1
sqrt( t(prop.browse) %*% vcv.psi %*% prop.browse )
# show missing part of variance 2 times cross-product of prop.browse * covariance
2*prod(prop.browse)*vcv.psi[1,2]
sqrt(sum(prop.browse^2*diag(vcv.psi))+2*prod(prop.browse)*vcv.psi[1,2])
```

`Whatsnew` *Summary of changes by version*

Description

A good place to look for changes. Often I'll add changes here but don't always get to it in the documentation for awhile. They are ordered from newest to oldest.

Details

Version 2.0.9 (1 Dec 2011)

- Patch was made to make.mark.model to fix bug in PIM creation for a multi-session model and there was just 1 session. Thanks to Erin Roche for helping to identify this bug.
- Patch was made to make.mark.model to fix bug in handling of mlogits for pi and Omega parameters with more than one group. These parameters were introduced with the RDMSMisClass and other new models that were recently added.
- Additional changes were made to export.MARK to re-fix changes for robust and nest survival model export to MARK.
- A function mark.wrapper.parallel written by Eldar Rakhimberdiev provides a parallel processing version of mark.wrapper. See the example in the help for the function. The parallel version is functionally the same and can be used in place of mark.wrapper to run sequentially or in parallel. It does not include the run argument however.
- A bug was fixed in get.real which caused an R error for a triangular PIM that had only a single entry. Thanks to Amanda Goldberg for discovering and reporting this error.

Version 2.0.8 (7 Oct 2011)

- Both setup.model and setup.parameters were re-written to use data files models.txt and parameters.txt to define models and parameters which should make it easier to add new models. The latter function is now much simpler and smaller.
- Model RDOccupEG now allows sharing Epsilon and Gamma parameters. Epsilon is the dominant parameter which gets the share=TRUE argument. See RDOccupancy for an example. Thanks to Jake Ivan for an example of what was needed.
- To avoid confusion, the arguments component and component.name were removed from the parameter specification because these have not been required since v1.3 when full support for individual covariates were included. Likewise the argument covariates in mark and make.mark.model was removed because it was only needed to support the component approach.
- export.MARK was modified so that if all individual covariates are output, it excludes factor covariates.
- Many more models were added to those supported in RMark. Now 92 of the 137 models in MARK are supported. See MarkModels.pdf in the RMark directory of your R library to see which models are supported (in red). Most of the remaining unsupported models are versions with mis-identification error and they are not shown in MarkModels.pdf.
- Previously RMark stored the input file in a temporary file Markxxx.tmp. Using a common filename caused problems when more than one model were spawned to different CPUs, so now it uses a random temporary file name. It also no longer uses common file markxxx.vcv. Thanks to Glenn Stauffer for testing these changes.
- Sessions are now labelled using value of session time rather than numerically from 1 to largest session number in robust designs. Thanks to Tommy Garrison for this suggestion.
- A bug was fixed in get.real which caused incorrect assignments of fixed parameter values in the unusual case where a fixed parameter had a non-zero design matrix row.
- mark.wrapper was modified so it returns a list of the models that were constructed if run==FALSE. Thanks to Eldar Rakhimberdiev for the suggestion and code.
- Code was added to extract.mark.output to extract deviance degrees of freedom. Thanks again to Eldar for contributing this code.

- A bug was fixed in `make.mark.model` that prevented use of sin link on within session parameters in a robust design model. Thanks to Tommy Garrison for reporting this bug.
- A bug was fixed in `make.mark.model` which prevented the use of time varying covariates with shared parameters. Thanks to Andre Breton for reporting this bug.
- simplify argument was removed from functions because I have not found a reason not to simplify and I have not been testing code with simplify=FALSE.

Version 2.0.7 (25 August 2011)

- Change to `make.mark.model` to fix bug in which mlogits were incorrectly assigned in ORDMS model when both Psi and pent used mlogit links. Thanks to Glenn Stauffer for identifying and tracking down this error.
- Fixed `export.chdata` and `export.MARK` so Nest survival models can be exported. Also changed default of argument ind.covariates to "all" which will use all individual covariates in the data in the file sent to MARK. Thanks to Jay Rotella for his help.
- Made change to `process.data` and `mark` to include a new argument reverse, which if set to TRUE with model="Multistratum" will reverse the timing of transition and survival. See `mstrata` for an example of a reverse multistratum model.
- Made change to `make.design.data` to allow for zero time intervals in non-robust design model. This was needed to allow use of the reverse time structure in multi-state models. In addition, for the reverse multistate model the function now adds an occasion (occ) field to the design data because the time field will be constant when with a 0 time interval. The row names of the design matrix and real parameters was extended for this model to include occasion (o) and occasion cohort (oc) to create unique labels because cohort and time are not unique with 0 time intervals.

Version 2.0.6 (1 July 2011)

- Change to MR resight examples so the output does not appear in notepad which was causing problem with check on CRAN submission

Version 2.0.5 (29 June 2011)

- Order of arguments for `model.average.marklist` were switched incorrectly such that ... was the second argument. This has been fixed to the original format where ... is at the end. This resulted in the value of any arguments other than the first to be ignored unless specifically named e.g., `parameter="Phi"`. Thanks to Rod Towell for reporting this error.
- Additional changes were made to .First.lib to 1) examine any MarkPath setting, 2) look in C:/program files/mark or c:/program files (x86)/mark, 3) or to search the path. If MARK executable cannot be found in any of those ways then a warning is issued that the user needs to set MarkPath to the path specification for the location of the MARK executable. Thanks to Bryan Wright for help with tracking down a bug.
- A bug in `add.design.data` was fixed where the function gave incorrect results in pim.type was anything other than "all". Thanks to Jeff Hostetler for reporting this error.
- The mark-resight models PoissonMR, LogitNormalMR, and IELogitNormalMR were added. This required some changes to `make.mark.model`,`make.design.data` and `compute.design.data` and the addition of an argument `counts` to `process.data` to provide mark-resight count data that are not in the capture history format. The format for `counts` is a named list of vectors

(one group) or matrices (each group is a row) where the names of the list elements are specific to the model. Currently there is no checking to make sure these are named correctly. Some of these models are very sensitive to starting values; thus the use of initial values in the examples. Thanks to Brett McClintock for his help incorporating these models.

Version 2.0.4 (1 June 2011)

- Change made to .First.lib to check for availability of mark software that depends on operating system. It now provides a warning rather than stopping package attachment. This allows the user to set MarkPath to some location outside of default location Program Files or Program Files (x86) without changing path which requires administrator privilege on Windows.
- Change made to `run.mark.model` to use shQuote because link to mark.exe was not working in some cases.

Version 2.0.3 (17 May 2011)

- Now requires R 2.13 to use path.package function
- Change made to .First.lib to check for availability of mark software that depends on operating system.

Version 2.0.2 (16 May 2011)

- MarkPath no longer needs to be set if mark.exe is no longer in the default location (c:/Program Files/mark) but is specified in the path. The code now uses the R function Sys.which to find the correct location for mark.exe, if it is contained in the Path.
- Code for crm models was removed from RMark and moved to a different R package called marked that is under-development. This removes the FORTRAN code and accompanying dll and some functions and help files that were extraneous to RMark capabilities. There is still some code in some functions for crm that could be removed at some point. None of this matters to those who use RMark for its original purpose as an interface to mark.exe.
- Many superficial changes were made to code so it could be posted on CRAN. Three changes that may be noticed by users involved renaming deriv.inverse.link, summary.ch and merge.design.covariates to deriv_inverse.link, summary_ch, and merge_design.covariates. These names conflicted with the generic functions deriv, summary and merge. It is only the last 2 that you may have in your scripts and you will have to rename them. The deprecated function merge.occasion.data was removed. Sorry for any inconvenience.
- The dependency on Hmisc for the examples was removed by replacing errbar with plotCI.

Version 2.0.1 (21 Feb 2011)

- Made a change to `run.mark.model` to handle output filenames that exceeded mark9999.
- Added an argument `prefix` in `mark`, `run.mark.model` and `cleanup`. Like other parameters for `mark`, it can also be used in `mark.wrapper` as one of the ... arguments. Previously the mark files have always been named "markннn.*". By specifying `prefix` you can now create sets of models with different prefixes. For example, `prefix="cu"` would result in cu001.*(cu001.out, cu001.inp, etc), cu002.* etc. This provides the ability to name files to do things like naming them based on the species being analyzed. In general, there is no need to work with these files directly because the filename.* is stored with each `mark` object and the various R functions use that link to provide the information from the files. If you use prefixes other than "mark", you'll

need to call call `cleanup` with each prefix to remove unused files. See `run.mark.model` for an example that shows use of the prefix argument to split the dipper data into separate analyses for each sex. Note that use of prefix was not mandatory here to separate the analyses but it provided a useful example.

- Additional usefulness has been coded for argument `initial` for assigning initial values to beta parameters. Previously, the options were either a vector of the same length as the new model to be run or a previously run model in a `mark` object from which equivalent betas are extracted based on their names in the design matrix. Now if the vector contains names for the elements they will be matched with the new model like with the model option for initial. If any betas in the new model are not matched, they are assigned 0 as their initial value, so the length of the `initial` vector no longer needs to match the number of parameters in the new model as long as the elements are named. The names can be retrieved either from the column names of the design matrix or from `rownames(x$results$beta)` where `x` is the name of the `mark` object.
- Using the feature above, I added a new argument `use.initial` to `mark.wrapper`. If `use.initial=TRUE`, prior to running a model it looks for the first model that has already been run (if any) for each parameter formula and constructs an `initial` vector from that previous run. For example, if you provided 5 models for p and 3 for Phi in a CJS model, as soon as the first model for p is run, in the subsequent 2 models with different Phi models, the initial values for p are assigned based on the run with the first Phi model. At the outset this seemed like a good idea to speed up execution times, but from the one set of examples I ran where several parameters were at boundaries, the results were discouraging because the models converged to a sub-optimal likelihood value than the runs using the default initial values. I've left this option in but set its default value to FALSE.
- A possibly more useful argument and feature was added to `mark.wrapper` in the argument `initial`. Previously, you could use `initial=model` and it would use the estimates from that model to assign initial values for any model in the set defined in `mark.wrapper`. Now I've defined `initial` as a specific argument and it can be used as above but you can also use it to specify a `marklist` of previously run models. When you do that, the code will lookup each new model to be run in the set of models specified by `initial` and if it finds one with the matching name then it will use the estimates for any matching parameters as initial values in the same way as `initial=model` does. The model name is based on concatenating the names of each of the parameter specification objects. To make this useful, you'll want to adapt to an approach that I've started to use of naming the objects something like p.1,p.2 etc rather than naming them something like p.dot, p.time as done in many of the examples. I've found that using numeric approach is much less typing and cumbersome rather than trying to reflect the formula in the name. By default, the formula is shown in the model selection results table, so it was a bit redundant. Now where I see this being the most benefit. Individual covariate models tend to run rather slowly. So one approach is to run the sequence of models (eg results stored in initial_marklist), including the set of formulas with all of the variables other than individual covariates. Then run another set with the same numbering scheme, but adding the individual covariates to the formula and using `initial=initial_marklist` That will work if each parameter specification has the same name (eg., p.1=list(formula=~time) and then p.1=list(formula=~time+an_indiv_covariate)). All of the initial values will be assigned for the previous run except for any added parameters (eg. an_indiv_covariate) which will start with a 0 initial value.
- I added a new function `search.output.files` to the set of utility functions. This can be useful to search all of the output files in a `marklist` for a specific string like "numerical

convergence suspect" or just "WARNING". The function returns the model numbers in the `marklist` that contain that string in the output file.

- Further changes were needed to `popan.derived` to handle data that are summarized (i.e. frequency of capture history >1). Thanks to Carl Schwarz for reporting and finding the change needed.
- A bug was fixed in `create.dm` was fixed so it would return a matrix instead of a vector with the formula ~1

Version 2.0.0 (14 Jan 2011)

- The packages msm, Hmisc, nlme, plotrix are now explicitly required to install RMark. These were used in examples or for specialty functions, but to avoid problems these must be installed as well.
- Added example for "CRDMS" model that was created by Andrew Paul. See `crdms`.
- Change was made for gamma link in v1.9.3 was only made to Pradel model and not Pradsen like it stated. Both now correctly use the logit link as the default for gamma. Thanks to Gina Barton for bringing this to my attention.
- Change was made in `make.mark.model` that prevented use of groups with Nest survival models. Thanks to Jeff Warren for bringing this to my attention.
- Change was made in `make.design.data` because re-ordering of parameters caused issues with the CRDMS model because the `subtract.stratum` were not being set for `Psi`.

Version 1.9.9 (2 Nov 2010)

- This version was built with R 2.12 and will not work with earlier versions of R. It contains both 32 and 64 bit versions and R will automatically ascertain which to use.
- Parameter ordering for some models (RDHet, RDFullHet, RDHHet, RDHFHet, OccupHet, RDOccupHetPE, RDOccupHetPG, RDOccupHetEG, MSOccupancy, ORDMS, and CRDMS) had to be changed such that the models could be imported into the MARK interface. This change can influence any code you have written for those models if you specified parameter indices because the ordering of the parameters were changed. For example, see change in example for `RDOccupancy` to use indices=c(1) from c(10). Thanks to Gary White for helping me work this out.
- Modified code in `extract.mark.output` to handle cases with more than 9999 real parameters because MARK outputs **** when it exceeds 9999.

Version 1.9.8 (15 Sept 2010)

- Added `model="CRDMS"` with parameters S, Psi, N, p, and c for closed robust design multi-state models
- Patched `popan.derived` which produced incorrect abundance estimates with unequal time intervals. Thanks to Andy Paul for finding this error and testing for me.
- Patched `export.MARK` which failed for robust design models. Thanks to Dave Hewitt and Gary White for discovering and isolating the problem.

Version 1.9.7 (14 April 2010)

- Added `model="Brownie"` with parameters S and f which is the Brownie et al. parameterization of the recovery model. "Recovery only" (in RMark) `model="Recovery"` which is also encounter type "dead" in MARK uses the Seber parameterization with parameters S and r which is also used in the models for live and dead encounter models.
- Added `model="MSLiveDead"` with parameters S, r, Psi and p. It is the multistate version of the Burnham model in which F=1.
- Added `compute.Sn` to utility functions for computation of natural survival from total survival when all harvest is reported. Patched `nat.surv` which was incorrectly rejecting based on model type.
- Added `var.components.reml` to provide an alternate variance components estimation using REML or maximum likelihood. It allows a random component that is not iid which is all that `var.components` can do.
- Replaced all T/F values with TRUE/FALSE to avoid conflicts with objects named TRUE or FALSE.

Version 1.9.6 (1 February 2010)

- Writing the `popan.derived` function has led me down all sorts of paths. I had to make one small change to this function to handle externally saved models. However, various changes listed below were brought on by using this function with a relatively large POPAN model.
- Most importantly I discovered an error in computations of real parameters with an mlogit link in which some of the real parameters involved in the mlogit link were fixed. This is NOT a problem if you simply used the real parameter values extracted from MARK; however, if you were using either `compute.real` (model.average uses this function) or `covariate.predictions` to compute those real parameters (with an mlogit link) then they may be incorrect. This would have been apparent because their value would have changed relative to the original values extracted from the MARK output. Correcting this error involved changes in `compute.real`, `convert.link.to.real` and `covariate.predictions` to correct the real parameter estimates and their standard errors. I've not found it in the MARK documentation but deduced that if you use the mlogit link and fix real parameters it uses those fixed real parameter values in the calculation. A simple example will make it clear. Consider pent for 5 occasions where the first is computed by subtraction and you then have 4 real parameters. Let's assume that the 3rd and fourth parameters were fixed to 0. Then the real parameters are calculated as follows: pent2=exp(beta2)/(1+ exp(beta2)+exp(beta3)+exp(0)+exp(0)), pent3=exp(beta3)/(1+ exp(beta2)+exp(beta3)+exp(0)+exp(0)), pent4=0, pent5=0 and pent1=1-pent2-pent3-0-0 (in this case). Obviously you would not want to fix any real parameters to be >1, <0 or to have the sum to be <0 or >1. This structure also had implications on how the standard error was calculated.
- In addition the coding was made more efficient in `covariate.predictions` for the case where `data(index=somevector)` is used without any data entries for covariate values in the design matrix. While the task performed with that use of the function could be done with `compute.real`, it is useful to have the capability in `covariate.predictions` as well because it will then model average over the listed set of parameters. The previous approach to coding was inefficient and led to very large matrices that were unnecessary and could cause failure with insufficient memory for large analyses.
- Calculation of NGross was added to `popan.derived` and logical arguments `N` and `NGross` were added to control what was computed in the call. In addition, argument `drop` was added

which is passed to `covariate.predictions` to control whether models are dropped when variance of betas are not all positive.

- `var.components` was modified to use qr matrix inversion. The returned value for beta is now a dataframe that includes the std errors which are extracted from the vcv matrix. Also, if the design matrix only uses a portion of the vcv matrix, the appropriate rows and columns are now extracted. Prior to this change, the standard errors would have been unreliable if the design matrix didn't use the entire set of thetas and vcv matrix.

Version 1.9.5 (4 December 2009)

- A bug in `covariate.predictions` was fixed that would assign fixed values incorrectly if the parameter indices were specified in anything but ascending order. The error would have been obvious to anyone that may have encountered it because estimated parameters would likely have been assigned a fixed value. In most cases indices would be passed in order if they were selected from the design data unless indices were chosen from more than one parameter type. I discovered it using the `popan.derived` function I added in v1.9.4 because it requests indices for multiple parameters in a single function call.

Version 1.9.4 (6 November 2009)

- Note that this version was built with R 2.10 which no longer supports compiled help files (chtml). If you were using compiled help files to get help with RMark, you'll need to switch to regular html files by using options(help_type="html") in R. You can put this command in your RProfile.site file so it is set up that way each time you start R. The functionality is the same but it is not as pretty. You can find the index (what used to be in a window on the left) as a link at the bottom of each help page.
- Changed `export.MARK` so it will not allow selection of a project name that would over-write an existing .inp file.
- Added the function `popan.derived` which for POPAN models computes derived abundance estimates by group and occasion and sum of group abundances for each occasion. For some reason RMark is unable to extract all of the derived parameters from the MARK binary file for POPAN models. This function provides the derived abundance estimates and their var-cov matrix and adds the abundance estimate sum across groups for each occasion which is not provided by MARK. Note that by default confidence intervals are based on a normal distribution to match the output of MARK, but if you want log-normal intervals use `normal=FALSE`.
- The `model.average.list` and `model.average.marklist` functions were modified to use revised estimator for the unconditional standard error (eq 6.12 of Burnham and Anderson (2002)) which is now the default in MARK. To use eq 4.9 (the prior formula) set the argument `revised=FALSE`.
- Fixed bug in `model.average.list` which in some cases failed when the list of var-cov matrices were specified.
- Code in `model.average.marklist` was changed to set standard error to 0 if the variance is negative. The same is done in the var-cov matrix for the variance and any corresponding covariances. The results from RMark will now match the model average results from MARK for this case. It is not entirely clear that this is the best approach when ill-fitted models are included.
- Changed code in `cleanup` to handle case in which a model did not run.

- Changed use of `grep` and `regexpr` in `convert.inp` and `extract.mark.output` to accomodate change in R.2.10. The code should work in earlier versions of R but if not update to R2.10.
- Created a function `adjust.value` and kept special case of `adjust.chat`. For any field other than `chat` it will adjust the value in `model$results`. As an example, to adjust the effective sample size (ESS) use `model.list=adjust.value("n",value,model.list)` where value is replaced with the ESS you want to use. As part of the change `model.table` was changed to recompute AICc.

Version 1.9.3 (24 September 2009)

- Default link function for Gamma in the Pradel seniority model had been incorrectly set to log and has now been changed to logit to restrict it to be a probability.
- A bug was fixed in `compute.real` and `covariate.predictions` in which confidence intervals were being incorrectly scaled by c (chat adjustment) instead of sqrt(c). The reported standard errors were correctly using sqrt(c) and only the confidence intervals for the real predictions were too large. Simply re-running the prediction computations for a model will provide the correct results. There is no reason to re-run the models. Also this in no way affects model selection.
- A related bug was fixed in `compute.real` and `covariate.predictions` which created invalid confidence intervals for real parameters if a probability link other than logit was used and a single type of link was used for all real parameters (e.g., sin).

Version 1.9.2 (10 August 2009)

- Added the function `export.MARK` which creates a .Rinp, .inp and optionally renames one or more output files for import to MARK. The July 2009 version of MARK now contains a File/RMARK Import menu item which will automatically create the MARK project using the information in these files. This prevents problems that have been encountered in creating MARK projects with RMark output because the data/group structures are setup exactly in MARK as they were in RMark. See `export.MARK` for an example and instructions.
- Fixed a problem in `make.design.data` which prevented use of `remove.unused` with unequal time intervals and more than one group.
- At least one person has encountered a problem with a very large number of parameters in which RMark created the input file with PIMs written in exponential notation for the larger indices. MARK will not accept that format and it will fail. The solution to this is to set the R option scipen to a positive number. Start with options(scipen=1) and increase if necessary.

Version 1.9.1 (2 June 2009)

- Fixed a problem `make.design.data` which was not using `begin.time` to label the session values
- Made a change in `export.chdata` like the change in `make.mark.model` to accomodate change with release of version R2.9.0.
- Made a change in `process.data` so that `strata.labels` can be specified for Multistrata designs like with ORDMS so an unobserved strata can be included.

- A warning was added to the help file for export.chdata and export.model so it is clear that the MARK database must be created correctly and with the .inp file created by export.chdata from the processed data that was used to create the models that are being exported. This is to ensure that the group structure is setup such that the assumed model structure for groups matches the model structure setup in the .inp file.

Version 1.9.0 (30 April 2009)

- Fixed a bug in summary.mark which occasionally produced erroneous results with showall=FALSE.
- Made a change in make.mark.model to accomodate change with release of version R2.9.0.
- RMark now requires R version 2.8.1 or higher.

Version 1.8.9 (9 March 2009)

- Changed model.average.marklist and covariate.predictions to set NaN or Inf results in v-c matrix to 0 to cope with poorly determined models. Also, for each function the dropping of models is now restricted to cases in which there are negative variances for the betas being used in the averaged parameter estimates. Unused betas are ignored. For example, if model.average is called with parameter="Phi", then the model will only be dropped if there is a negative variance for one of the betas associated with "Phi".
- In var.components the tolerance value (tol) in the call to uniroot was reduced to 1e-15 which should provide better estimates of the process variance when it is small. Previously a process variance less than 1e-5 would be treated as 0.
- Made changes to cleanup, coef.mark, and make.mark.model to accomodate externally saved model objects (external=TRUE).

Version 1.8.8 (5 December 2008)

- An error was fixed in make.time.factor which created incorrect assignments when only some of the time dependent variables contained a "." for occasions with no data.
- Patched compute.design.data which was not creating the design data in the same order as the PIM construction for the newly added ORDMS model.
- Generalized section of code in make.mark.model to handle mlogit structure for ORDMS model.
- Fixed a bug in process.data in which the initial ages were not correctly assigned in some situations with multiple grouping variables. Note that it is always a good idea to examine the design data after it is created to make sure it is structured properly because it relates the data and model structure via the grouping variables and the pre-defined variables (ie age, time etc), While I've done a lot of testing, I have certainly not tried every possible example and there is always the potential for an error to occur in a circumstance that I've not encountered.

Version 1.8.7 (13 November 2008)

- An argument common.zero was added to function make.design.data and compute.design.data. It can be set to TRUE to make the Time variable have a common time origin of begin.time which is useful for shared parameters like p and c in closed capture and similar models.
- The function read.mark.binary was patched to work with the newer versions of MARK.EXE since 1 Oct 2008.
- The model type ORDMS for open robust design multi-state models was added. An example data set will be added at a later date after further testing has been completed.

- Some patches were made to fix some aspects of profile intervals and to fix adjustment by chat in summary.mark when showall=F. The notation for profile intervals is now included in the field model$results$real$note where model is the name of a mark model. Previously an incomplete notation was kept in model$results$real$fixed but that field is now used exclusively to denote fixed parameters. It is important to realize that profile intervals computed by MARK are only found in model$results$real$note and are not changed by a chat adjustment unless the model is re-run. None of the intervals computed by RMark and displayed by summary.mark are profile intervals.

Version 1.8.6 (28 October 2008)

- A bug in an error message for initial.ages in process.data was fixed.
- A new function var.components was added to provide variance components capability as in the MARK interface except that shrinkage estimators are not computed currently.
- Fixed parameter values are now being reported correctly by covariate.predictions. Also over-dispersion (c>1) was not being included in the variances for parameters except those using the mlogit link.
- Some utility functions were added including pop.est,nat.surv, and extract.indices.
- The function model.average has been changed to a generic function. Currently it supports 2 classes: 1) list, and 2) marklist. The latter was the original model.average which has been renamed model.average.marklist and the first argument has been renamed x instead of model.list to match the standard generic function approach. The previous syntax model.average(...) will work as long as the usage does not name the first agument as in the example model.average(model.list=dipper.results,...). The list formulation (model.average.list) was created to enable a generic model averaging of estimates instead of just real parameter estimates from a mark model. It could be used with any set of estimates, model weights and estimates of precision.
- A change is needed to read.mark.binary to accomodate the change to mark.exe with the version dated 1 Oct 2008. Some data types (notably Nest survival) may not work with the new version of mark.exe. Working with Gary to make the patch. If you need an older version of mark.exe contact me.

Version 1.8.5 (8 October 2008)

- A bug in process.data was fixed that prevented use of a dataframe contained in a list while using the groups argument.
- Profile intervals on the real parameters can now be obtained from MARK using the arguments profile.int and optionally chat in mark. The argument profile.int can be set to TRUE and a profile interval will be constructed for all real parameters, or a vector of parameter indices can be specified to restrict the profiling to certain parameters. The value specified by chat is passed to MARK for over-dispersion.
- References to cjs, js etc have been removed from here because this code was removed 5/11/11.
- Yet another fix to summary.ch which gave incorrect results for the number recaptured at least once when marray=F and the data contained non-unity values for freq.

Version 1.8.4 (29 August 2008)

- A generic function coef.mark was added to extract the table of betas from the model with the expression coef(model) where model is a mark model that has been run and contains output. The table includes standard errors and confidence intervals.
- An argument brief was added to summary.mark. If brief=TRUE the real parameters are not included in the summary.
- References to cjs, js etc have been removed from here because this code was removed 5/11/11.
- A bug in summary.ch was fixed. It would produce erroneous results when the data contained a non-constant freq field. Results with the default of freq=1 were fine.
- The function adjust.chat and its help file were changed such that it was clear that a model.list argument was needed.

Version 1.8.3 (25 July 2008)

- For robust design models, an error trap was added to process.data to make sure that the capture history length matches the specification for the time.intervals. This error was already trapped for non-robust models.
- Fixed an error in make.mark.model that prevented interaction model of session/time-specific individual covariates in a robust design model.
- Fixed an error in process.data so that the field freq is optional for nest survival data sets.
- print.mark was modified to add an argument input which if set to input=TRUE will have the MARK input file be displayed rather than the output file. Also, wait=FALSE was set in the system command which means the viewer window will be opened and you can carry on with R. Before you had to close the viewer window before proceeding with R.
- An example RDOccupancy provided by Bret Collier was added for the Robust Occupancy model which shows the use of session and time-varying individual covariates in a robust design model.

Version 1.8.2 (26 June 2008)

- summary.ch was modified to allow missing cohorts (no captures/recaptures) for an occasion and to fix a bug in which bygroup=FALSE did not work when groups were defined.
- To avoid running out of memory, an argument external has been added to collect.models, mark, rerun.mark, and run.mark.model. As with all arguments of mark, external can also be set in mark.wrapper. Likewise, external can also be set in run.models and it is passed to run.mark.model. The default is external=FALSE but if it is set to TRUE then the mark model object is saved in an external file with an extension .rda and the same base filename as its matching MARK output files. The mark object in the workspace is a character string which is the name of the file with the saved image (e.g., "mark001.rda"). If external=TRUE with mark.wrapper then the resulting marklist contains a list entry for each mark model which is only the filename and then the last entry is the model.table. All of the functions recognize the dual nature of the mark object (i.e., filename or mark object) in the workspace. So even if the mark object only contains the filename, functions like print.mark or summary.mark will work. However, if you have used external=TRUE and you want to look at part of a mark object without using one of the functions, then use the function load.model. Whereas, before you may have typed mymark$results, if you use external=TRUE, you would replace the above with load.model(mymark)$results.

- Functions `store` and `restore` were created to `store` externally and `restore` models from external storage into the R workspace. They work on a `marklist` and are only needed to `store` externally existing marklist models or ones originally created with `external=FALSE` or to `restore` if you change your mind and decide to keep them in the R workspace.
- Error in setup for robust design occupancy models with more than 2 primary sessions was fixed. The error resulted in mark.exe crashing.
- The concept of time-varying individual covariates has been expanded to include robust design models which have both primary (session) and secondary (time) occasion-specific data. For a robust design, a time-varying individual covariate can be either session-dependent or session-time dependent. As an example, if there are 3 primary sessions and each has 4 secondary occasions, then the individual covariates can be named x1,x2,x3 to be primary session-dependent or named x11,x12,x13,x14,x21,x22,x23,x24,x31,x32,x33,x34. The value of x can be any name for the covariate. In the formula only the base name is used (e.g., `~x`) and RMark fills in the individual covariate names that it finds that match either the session or session-time individual covariates.

Version 1.8.1 (19 May 2008)

- Added function `summary.ch` to provide summaries of the capture history data (resighting matrices and m-arrays). It will not work with all types of models at present. It will work with CJS and Jolly-type models.
- Added argument `model.name` to `model.table` to be able to use alternate names in the model table. It can use either the model name with each mark object which uses a formula notation (the current approach) or it can use the name of the R object containing the mark model (`model.name=FALSE`). See `model.table` for an example. Also, the help file for `model.table` was updated to reflect the code changes implemented in version 1.7.3.
- Code in `mark.wrapper` was modified to output the number of columns and column names of the design matrix for each model if `run=FALSE`. This allows a check of each of the columns included in the model. By reviewing these you can assess whether the model was constucted as you intended. If there is any question you can either use `model.matrix` or `make.mark.model` to examine the design matrix more thoroughly.
- A bug in the new function `merge.design.covariates` was fixed in which `merge` was sorting the design data which does obvious bad things. Adding `sort=FALSE` does not appear to mean that the data frame is left in its original order. To prevent this, the dataframe is forced to remain in its original order by adding a sequence field for re-sorting after the merge.

Version 1.8.0 (8 May 2008)

- Fixed a bug in `model.average` that caused it to fail and issue an error when any of the models included a time dependent covariate in the parameter being averaged.
- Added `merge.design.covariates` which is meant to replace `merge.occasion.data`. This new function allows covariates to be assigned by `time`, `time` and `group`, or just `group`. It also uses a simplified list of arguments and works with individual design dataframes rather than the entire ddl. It uses the R function `merge` which can be used on its own to merge design covariates into the design data. You can use `merge` directly as this function only checks for some common mistakes before it calls `merge` and it handles reassignemnt of row names in the case were design data have been deleted. An example, where you might want to use `merge` instead of this function would be situations where the design data are not just group, time or

group-time specific. For example, if groups were specified by two different factor variables say initial age and region and the design covariates were only region-specific. It would be more efficient to use `merge` directly rather than this function which would require an entry for each group which would be each pairing of inital age and region. If you use `merge` and you deleted design data prior to merging, save the row.names, merge and then reassign the row.names.

- An argument `run` was added to `mark.wrapper`. If set to FALSE, then it will run through each set of models in `model.list` and try to build each model but does not attempt to run it. This is useful to check for and fix any errors in the formula before setting off a large run. If you use `run=FALSE` do not include arguments that are meant to be passed to `run.mark.model` like `adjust`.

Version 1.7.9 (7 April 2008)

- `make.design.data` was fixed so that `remove.unused=T` will work properly when different `begin.time` values are specified for each group.

Version 1.7.8 (12 March 2008)

- Changed the default link for N to log in the `setup.parameters` for the `HetClosed` and `FullHet`. It was incorrectly set to logit which created incorrect estimates to be computed in `model.average` because MARK forces the log link for N regardless of what is set in the input file.

Version 1.7.7 (6 March 2008)

- Supressed warning message that occurred with code to check the validity of the sin link in `make.mark.model`.
- Fixed a couple of bugs in `covariate.predictions` that prevented it from working for some cases after including code for the sin link.
- Added function `release.gof` to construct the RELEASE goodness of fit test and extract the TEST2 and TEST3 final chi-square, df and P-values.

Version 1.7.6 (26 Feb 2008)

- `make.mark.model` was modified to change the capitalization of the link functions and to remove all spaces after "=" in the input file for mark.exe. These differences were preventing the MARK interface from fully importing the model. Although the model would be imported and could be run inside the MARK interface, median c-hat would not run and would give an error stating "Invalid Link" for any model imported from RMark. Now transfering a model from RMark to the MARK interface is fully functional (I hope). If you want to import an output file that was created with a prior version of RMark without re-running it, use a text editor on the output file and remove any spaces before and after an = sign. Then change the capitalization of the links to "Logit", "MLogit", "Log", "LogLog", "CLogLog", "Identity".
- The sin link is now supported if the formula for the parameter generates an identity matrix for the parameter. For example, if you use ~-1+time instead of ~time then the resulting design matrix will be an identity for time. Likewise, for interactions use ~-1+group:time instead of ~group*time. If you select the sin link and the resulting design matrix is not an identity for the parameter, an error will be given and the run will stop.

- To match the output from MARK, the confidence intervals for real parameters using any 0-1 link including loglog,cloglog,logit and sin are now computed using the logit transformation. For previous versions this will only affect any results that were using loglog and cloglog. Previously, it was using the chosen link to compute the se and the interval endpoints. The latter is still used for the log and identity links which are not bounded in 0-1.
- The model "Jolly" was added to the supported list of models. Parameters include Phi,p,Lambda,N. It is not a particularly numerically stable model and often will not converge. Use of options="SIMANNEAL" in call to `mark` is recommended for better convergence. It will take much longer to converge but is mroe reliable.

Version 1.7.5 (24 Jan 2008)

- `model.average` was modified to ignore any models that did not run and either had no attached output file or no results.
- `read.mark.binary` and `extract.mark.output` were modified to extract and store the real.vcv matrix (var-cov matrix of the simplified real parameters) in the mark object if realvcv=TRUE. The default is realvcv=FALSE. This argument has been added to functions `mark`, `run.mark.model` and `rerun.mark`.
- An argument delete has also been added to `mark` and `run.mark.model`. The default value is FALSE but if set to TRUE it deletes all output files created by MARK after extracting the results. This is most useful for simulations that could easily create thousands of output files and after extracting the results the model objects are no longer needed. This is just a convenience to replace the need to call `cleanup`.

Version 1.7.4 (10 Jan 2008)

- A bug in `make.mark.model` was fixed. It was preventing creation of individual (site) covariate models for parameters with only a single parameter (single index) in certain circumstances like Psi1 in the MSOccupancy model.
- The fix to `merge.occasion.data` in version 1.7.1 did not work when design data had been deleted. That has been remedied.
- Various functions with some operating specific calls have been modified so they will work on either Windows or Linux. Thus, the there is a single file for all source/help for both operating systems in RMarkSource.zip. It can be downloaded to either Windows or Linux to build the package. You need to build the package for Linux but not for Windows. For Windows, you only need RMark.zip which contains the pre-built package which only needs to be installed. Currently, with Linux the variable MarkPath is ignored and mark.exe is assumed to be in the path. Also, for Linux the default for MarkViewer is "pico" (an editor on some Linux machines). This can be modified in `print.mark` or by setting MarkViewer to a different value. The one Linux specific function is `read.mark.binary.linux`. The function `extract.mark.output` calls either `read.mark.binary.linux` or `read.mark.binary` depending on the operating system. A Linux version of mark.exe (32 or 64 bit) can be obtained from Evan.

Version 1.7.3 (4 Jan 2008)

- In working with the occupancy models, it became apparent that it would be useful to have a new function called `make.time.factor` which creates time-varying dummy variables from a time-varying factor variable. An example is given using observer with the occupancy dataset `weta` from the MacKenzie et al Occupancy modelling book.

- To match the results in the book, I added arguments `use.AIC` and `use.lnl` to function `model.table` to construct a results table with AIC rather than AICc and -2LnL values. The latter is more useful with a mix of models some using individual covariates and others not.
- A modification was made to `make.mark.model` with the `MSOccupancy` model to fix the name of the added data for parameter `p1` when `share=TRUE` to be `p2`. For an example which uses p2 to construct an additive model, see `NicholsMSOccupancy`.

Version 1.7.2 (20 Dec 2007)

- In changing code for the occupancy models, a brace was misplaced which prevented the nest survival models from working. This has been fixed. Also, the example code for `mallard` and `killdeer` was modified to exclude the calls to process the input file. This enables use of the function `example()` to run the example code (e.g. `example(mallard)`). From now on as I add examples they are being included in my test set to avoid this type of problem in the future.

Version 1.7.1 (14 Dec 2007)

- If you update with this version of RMark make sure to update MARK also, so you get the fixes for some of the occupancy models.
- A minor bug was fixed in function `merge.occasion.data` that created duplicate row names and prevented the design data from being used in a model.
- Thirteen different occupancy models were added. Models in the following list use the designation from MARK: `Occupancy, OccupHet, RDOccupEG, RDOccupPE,RDOccupPG, RDOccupHetEG, RDOccupHetPE, OccupRNNegBin,OccupRPoisson,OccupRNegBin,MSOccupancy`. `Het` means it uses the Pledger mixture and those with `RD` are the robust design models. The 2 letter designations for the RD models are shorthand for the parameters that are estimated. For `EG`, Psi, Epsilon, and Gamma are estimated, for `PE` gamma is dropped and for `PG`, Epsilon is dropped. For the latter 2 models, Psi can be estimated for each primary occasion. The last 5 models include the Royle/Nichols count (RPoisson) and presence (RNPoisson) models and the multi-state occupancy model. See `salamander` for an example of `Occupancy, OccupHet`, `Donovan.7` for an example of `OccupRNPoisson,OccupRNNegBin`, `Donovan.8` for an example of `OccupRPoisson,OccupRNegBin`, see `RDSalamander` for an example of the robust design models and `NicholsMSOccupancy` for an example of `MSOccupancy`. `salamander` data.
- The functions `create.model.list` and `mark.wrapper` were modified so that a list of parameters can be used to loop. This is useful in the situation with shared parameters such as `p1` and `p2` in the `MSOccupancy` model, `closed` models etc. See `p1.p2.different.dot` in `NicholsMSOccupancy` for an example. It can also be useful if the model definitions are linked conceptually (e.g., when one parameter is time dependent, the other should also be time dependent).
- The "." value in an encounter history is now acceptable to RMark and gets passed to MARK for interpretation as a missing value.
- `print.marklist` was fixed to show the model table properly after a c-hat adjustment was made. The change in the code in version 1.6.5 to add parameter specific values to the model table had the side-effect of dropping the model name if c-hat was adjusted.

Version 1.7.0 (7 Nov 2007)

- A function `deltamethod.special` for computation of delta method variances of some special functions was added. It uses the function `deltamethod` from the package `msm`. You need to install the package `msm` from CRAN to use it.

- A more complete example (mallard) created by Jay Rotella was added for the nest survival model. His script provides a nice tutorial for RMark and the utility of R to provide a wide-open capability to calculate/plot etc with the results. It also demonstrates the advantages of scripting in R to document your analysis and enable it to be repeated. Before you use his tutorial you need to install the package plotrix from CRAN. At a later date, Jay has said he will add some additional examples to demonstrate use of the deltamethod function to create variances for functions of the results from MARK.
- Various changes were made to help files. A more complete description of cleanup was given to tie into mallard example.

Version 1.6.9 (10 Oct 2007)

- Nest survival model was added to list of MARK models supported by RMark. See killdeer for an example. Note that the data structure for nest models is completely different from the standard capture history so the functions import.chdata, export.chdata and convert.inp do not work with nest data structure.
- Slight change was made to run.mark.model and print.markto accomodate change in R 2.6.0.

Version 1.6.8 (2 Oct 2007)

- Changes were made to merge.occasion.data to enable group and time-specific design covariates to be added to the design data.
- Change was made to setup.parameters to use a log-link for N in the closed-capture models. MARK forces that link for N but the change was needed for model.average which does the inverse-link computation. Note that the reported N in model.average is actually f0 (number not seen). To get the correct values for N simply add M_t+1 (unique number captured) to f0. That is the way MARK computes N. The std error and confidence interval is on f0 such that the lower ci on N will never be less than M_t+1.
- An error was fixed in the output of model.average. When you selected a specific parameter, it was giving a UCL which was a copy from one of the models and not the UCL from the model averaging. If you didn't specify vcv=T it only showed the errant UCL and if you did specify vcv=T then it showed the correct LCL and UCL but then added the errant UCL in a column. This occurred because it was adding covariate data for the specific parameter and was shifted a column because of a change in 1.6.1.

Version 1.6.7 (7 Aug 2007)

- Changes were made in print.mark, print.summary.mark and compute.design.data to acommodate changes in V2.5.1 of R. When upgrading versions of R problems may occur if RMark was built with an earlier version of R. The version of R that was used to build RMark is listed on the screen each time it is loaded with library(RMark) `This is RMark 1.6.7 Built: R 2.5.1; i386-pc-mingw32; 2007-08-07 09:00:33; windows`
- The help file for import.chdata was expanded to clarify the differences between it and convert.inp and the use of the freq field.

Version 1.6.6 (14 May 2007)

- Function make.mark.model was fixed so that the real label indices were properly written when simplify=FALSE is used.

- Function `make.mark.model` was also changed to remove the parameter simplification for `mlogit` parameters that was added in v1.4.5. I mistakenly assumed that the `mlogit` parameters were setup such that the normalization to sum to 1 was done will all the real parameters in the set (i.e., all PSI for a single stratum). In fact, the `mlogit` values are only specified for the unique real parameters so if there is any simplification and the sum of the probabilities is close to 1 (excluding subtraction value) the values will not be properly constrained. For example, with the `mstrata` data if the problem was constrained such that PSI from AtoB was equal to AtoC, it is still necessary to have these as separate real parameters and constrain them with the design matrix. As it turns out, with the `mstrata` example it does not matter because the problem is such that the sum of Psi for AtoB and AtoC is not close to 1 (same for other strata) and any link will work. This change will only be noticeable in situations in which the constraint matters (i.e., the probability for the subtraction parameter is near 0). The change back to non-simplification for `mlogit` parameters may increase execution times because the design matrix size has been increased. Previous users of the Multistrata design will see very little difference in there results if they only used models containing stratum:tostratum because that will create an all-different PIM within each `mlogit` set. When I ran the `mstrata` examples with this version and compared them to v1.6.5 the results were different but they were differences in the 5th or smaller decimal point due to differences in numerical optimization.

Version 1.6.5 (3 May 2007)

- Function `model.table` was modified to include parameter formula fields in the `model.table` dataframe of a `marklist`. Previously only the `model.name` was included which is a concatenation of the individual parameter formulas. The additional fields allows extracting the model table results based on one of the parameter formulas or to create a matrix of model AICc or other values with rows as one parameter and colums as the other. See `model.table` for an example.
- Function `process.data` was modified such that factor variables used for grouping retain the ordering of the factor levels in the data file. Previously they would revert back to default ordering and the re-leveling would also have to be repeated on the design data also.
- An argument `brief` was added to `mark` to control amount of summary output.
- Fixed a bug in `get.real` that prevented computation for models without the stored covariate values.
- Added code to `make.mark.model` that prevents constructing models with empty rows in the design matrix unless the parameter is fixed. For example, if you were to try ~-1+Time for the dipper data, it will fail now because there is no value for the intercept (Time=0).
- Function `mark.wrapper` outputs the model name to the screen before running the model which helps associate any error messages to the model if `output=F`.

Version 1.6.4 (7 March 2007)

- A new function `export.model` was created to copy the output files into the naming convention needed to append them into a MARK .dbf/.fpt database so they can be used with the MARK interface features. This is useful to be able to use some of the features not contained in RMark such as median c-hat and variance component estimation. To create a MARK database, first use `export.chdata` to create a .inp file to pass the data into MARK. Start MARK and use File/New to create a new database. Select the appropriate Data Type (model in RMark) and fill in the appropriate values for encounter occasions etc. For the Encounter Histories File

Name, select the file you created with `export.chdata`. Once you have created the database in the Program MARK interface, click on the Browse menu item and then Output/Append and select the output file(s) (i.e those with a Y.tmp) that you exported with `export.model`. Note that this will not work with output files run with versions of RMark prior to this one because the MARK interface will give a parse error for the design matrix. To get around that you can edit the output file and remove the spaces in the line with the design matrix header. For example, it should look as follows `design matrix constraints=7 covariates=7` without spaces around the = sign.

- The minor change described above was made in the input file with spacing on the design matrix line to enable proper appending of the output into a MARK .dbf/.fpt database.
- The function `cleanup` was modified to delete all mark*.tmp files. Do not use `cleanup` until you have appended any exported models.
- An argument `use.comments` was added to `import.chdata` to enable comment fields to be used as row names in the data frame. A comment is indicated as in MARK with /* comment */. They can be anywhere in the record but they must be unique and they can not have a column header (field name).
- Function `create.model.list` was modified such that it only includes lists with a `formula` element. This prevents collecting other objects that are named similarly but are not model defintions.

Version 1.6.3 (5 March 2007)

- A minor change in `make.mark.model` and `find.covariates` was made to accommodate use of the same covariate in different formulas (e.g. Phi and p). Previous code worked except any call to `get.real` would fail. Previously a duplicate of the covariate was entered in the data file to MARK. Now only a single copy is passed.
- An argument `default` has been added to the model definition (`parameters` in `make.mark.model` and `model.parameters` in `mark`). The argument sets the default value for parameters represented by design data that have been deleted.
- Checks were added in `make.mark.model` to fail if any of the individual covariates used are either factor variables or contain NAs. Both could fail in the MARK.EXE run but the error message would be less obvious. Factor variables can work as an individual covariate, if the levels are numeric. But it was easier to exclude all factor variables from being individual covariates. They can easily be converted to a continuous version (e.g. Blackduck$BirdAge=as.numeric(Blackduck$BirdAge)-1). The code for the `Blackduck` was changed to make BirdAge a continuous rather than factor variable. Factor variables can still be used to define groups and then used in the formula. They just can't be used as individual covariates. This change was made because a factor variable was in the data but not defined in `groups` and when it was used in the formula it would create a float error in MARK.EXE and that would be confusing and hard to track down.

Version 1.6.2 (28 Feb 2007)

- The fix in 1.6.1 to avoid the incorrect design matrix was not sufficiently general and created a parse error in R if you attempted to use any design data covariates that were created with a cut function to create factor variables by binning a variable. This has been corrected in this version.
- The code in `read.mark.binary` has been changed to skip over the v-c matrix for the derived parameters if it is not found in the file. This was causing an error with the PRADREC model type.

Version 1.6.1 (17 Jan 2007)

- An important bug was fixed in `make.mark.model` in which an incorrect design matrix would be created if you used two individual covariates in the same formula whereby one of the covariate names was contained within the other. For example, if you used ~mass+mass2 where mass2=mass^2, it would actually create a design matrix with columns mass product(mass,mass2) which would be the model mass+mass^3. This happened due to the way the code identified columns where it needed to replace dummy values with individual covariate names. Since mass was contained in mass2 it added mass to the column as a product. The code now does exact matching so the error can no longer occur.
- An argument `indices` was added to the function `model.average` which enables restricting the model averaging to a specific set of parameters as identified by the all-different parameter indices. This is most useful in large models with many different indices such that memory limitations are encountered in constructing the variance-covariance matrix of the real parameters. For example, with a CJS analysis of data with 18 groups and 26 years of data, the number of parameter indices exceeds 22,000. Even by restricting the parameters to either Phi or p with the `parameter` argument there are still 11,000 which would attempt to create a matrix containing 11,000 x 11,000 elements which can exceed the memory limit. In most cases, there are far fewer unique parameters and this argument allows you to select which parameters to average.
- Time-varying covariates are no longer needed for all times if the formula is correctly written to exclude them in the resulting design matrix. `make.mark.model` still reports missing time-varying covariates but will continue to try and fit the model but if the missing variables are used in the design matrix the model will fail. As an example consider a time varying covariate x for recapture times 1990 to 1995. The code expects to find variables x1990, x1991, x1992, x1993, x1994, x1995. However, lets say that the values are only known for 1993-1995. If you define a variable I'll call recap in the design data which has a value 1 for 1993-1995 and a value 0 for 1990-1992 then if you use the formula ~recap:x the resulting design matrix will only use the known variables for 1993-1995 but you will still be warned that the other values (x1990 - x1992) are missing.
- A bug was fixed in `extract.mark.output` which prevented it from obtaining more than the last mean covariate value from the MARK output.
- `fill.covariates` was modified such that only a partial list of covariate values need to be specified with `data` and the remainder are filled in with default values depending on argument `usemean`.
- The output from `summary.mark` was modified for real parameters when `se=T` to include `all.diff.index` to provide the indices of each real parameter in the all-different PIM structure. They are useful to restrict `covariate.predictions` and `model.average` to a specific set of real parameters.
- A new function `covariate.predictions` was created to compute real parameter values for multiple covariate values and their variance-covariance matrix. It will also model average those values if a marklist is passed to the function. Two examples from chapter 12 of Cooch and White are provided to give examples of models with individual covariates and the use of this function.
- The default value of `vcv` in `model.average` has been changed to `FALSE`.

Version 1.6.0 (27 Nov 2006)

- A bug was fixed in `PIMS` which prevented it from working with Multistrata models.
- Bugs were fixed in `make.design.data` which prevented use of argument `remove.unused=T` with Multistrata models and also for any type of model when there were no grouping variables.
- Bugs were fixed in `process.data` which gave incorrect ordering of intial ages if the factor variable for the age group was numeric and more than two digits. Also, the number of groups in the data was not correct if the number of loss on capture records exceeded the number without loss on capture within a group.
- Bugs were fixed in `setup.parameters` and `setup.model` that prevented use of the Barker model and that reported an erroneous list of model names when an incorrect type of model was selected.

Version 1.5.9 (26 June 2006)

- A bug was fixed in `convert.inp` which prevented the code from working with groups and two or more covariates. Note that there are limitations to this function which may require some minor editing of the file. The limitations have been added to the help file (`convert.inp`).

Version 1.5.8 (22 June 2006)

- Argument `options` was added to `mark` and `make.mark.model` with a default NULL value. It is simply a character string that is tacked onto the `Proc Estimate` statement for the MARK .inp file. It can be used to request options such as NoStandDM (to not standardize the design matrix) or SIMANNEAL (to request use of the simulated annealing optimization method) or any existing or new options that can be set on the estimate proc.
- A bug in `model.table` was fixed so it would accomodate the change from v1.3 to a marklist in which the model.table was switched to the last entry in the list.
- A bug in `summary.mark` was fixed so it would properly display QAICc when chat > 1.
- Function `adjust.chat` was modified such that it returns a marklist with each model having a new chat value and the model.table is adjusted for the new chat value.
- Function `adjust.parameter.count` was modfied so it returns the mark model object rather than using eval to modify the object in place. The latter does not work with models in a marklist and calls made within functions.

Version 1.5.7 (8 June 2006)

- Argument `data` was added to function `model.average` to enable model averaging parameters at specific covariate values rather than the mean value of the observed data. An example is given in the help file.
- Argument `parameter` of function `model.average` now has a default of NULL and if it is not specified then all of the real parameters are model averaged rather than those for a particular type of parameter (eg p or Psi).
- A bug was fixed in function `compute.real` that caused the function to fail for computations of `Psi`.

Version 1.5.6 (6 June 2006)

- `print.summary.mark` was modified so fixed parameters are noted.

- Argument show.fixed was added to summary.mark to control whether fixed parameters are shown as NA (FALSE) or as the value at which they were fixed. If se=T the default is show.fixed=T otherwise show.fixed=F. The latter is most useful in displaying values in PIM format (without std errors), so fixed values are displayed as blanks instead of NA.
- Argument links was added to convert.link.to.real and the default value for argument model is now NULL. One or the other must be given. If the value for links is given then they are used in place of the links specified in the model object. This provides for additional flexibility in changing link values for computation (eg use of log with mlogit).
- Argument drop was added to model.average. If drop=TRUE (the default), then any model with one or more non-positive (0 or negative) variances is not used in the model averaging.
- An error in computation of the v-c matrix of mlogit link values in compute.links.from.reals was fixed. This did not affect confidence intervals for real parameters (eg Psi) in model.average because it uses the logit transformation for confidence intervals on real parameters that use mlogit link (eg Psi).
- get.real was unable to extract a single parameter value(eg constant Phi model). This was fixed.
- The argument parm.indices was removed from the functions compute.real and convert.link.to.real because the subsetting can be done easily with the complete results returned by the functions. This changed the examples in fill.covariates.
- compute.real and subsequently get.real return a field fixed when se=TRUE that denotes whether a real parameter is a fixed parameter or an estimated parameter at a boundary which is identified by having a standard error=0.

Version 1.5.5 (1 June 2006)

- model has been deleted from the arguments in TransitionMatrix. It was only being used to ascertain whether the model was a Multistrata model. This is now determined more accurately by looking for the presence of tostratum in the argument x which is a dataframe created for Psi from the function get.real. The function also works with the estimates dataframe generated from model.average. See help for TransitionMatrix for an example.
- An argument vcv was added to function model.average. If the argument is TRUE (the default value) then the var-cov matrix of the model averaged real parameters is computed and returned and the confidence intervals for the model averaged parameters are constructed. Models with non-positive variances for betas are reported and dropped from model averaging and the weights are renormalized for the remaining models.
- A new function compute.links.from.reals was added to the library to transform real parameters to its link space. It has 2 functions both related to model averaged estimates. Firstly, it is used to transform model averaged estimates so the normal confidence interval can be constructed on the link values and then back-transformed to real space. The second function is to enable parametric bootstrapping in which the error distbution is assumed to be multivariate normal for the link values. From a single model, the link values are easily constructed from the betas and design matrix so this function is not needed. But for model averaging there is no equivalent because the real parameters are averaged over a variety of models with the same real parameter structure but differing design structures. This function allows for link values and their var-cov matrix to be created from the model averaged real estimates.

Version 1.5.4 (30 May 2006)

- In function `mark` an argument `retry` was added to enable the analysis to be re-run up to the number of times specified. An analysis is only re-run if there are "singular" beta parameters which means that they are either non-estimable (confounded) or they are at a boundary. Beginning with this version, `extract.mark.output` was modified such that the singular parameters identified by MARK are extracted from the output (if any) and the indices for the beta parameters are stored in the list element `model$results$singular`. The default value for `retry` is 0 which means it will not retry. When the model is re-run the initial values are set to the values at the completion of the last run except for the "singular" parameters which are set to 0. Using `retry` will not help if the parameters are non-estimable. However, if the parameters are at a boundary because the optimization "converged" to a sub-optimal set of parameters, then setting `retry` to 1 or a suitably small value will often help it find the MLEs by moving away from the boundary. If the parameters are estimable and setting `retry` does not work, then it may be better to set new initial parameters by either specifying their values or using a model with similar parameters that did converge.
- A new function `rerun.mark` was created to simplify the process of refitting models with new starting values when the models were initially created with `mark.wrapper` which runs a list of models by using all combinations of the formulas defined for the various parameters in the model. Thus, individual calls to `mark` are not constructed by the user and re-running an analysis from the resulting list would require constructing those calls. The argument `model.parameters` is now stored in the model object and it is used by this new function to avoid constructing calls to rerun the analysis. With this new function you only need to specify the model list element to be refitted, the processed dataframe, the design data and the model list element (or different model) to be used for initial values. See `rerun.mark` for an example.
- To make `rerun.mark` a viable approach for all circumstances, the functions `mark.wrapper` and `model.table` were modified such that models that fail to converge at the outset (i.e., does not provide estimates in the output file) are stored in the model list created by the former function and they are reported as models that did not run and are skipped in the model.table by the latter function. This enables a failed model to be reanalyzed with `rerun.mark` using another model that converged for starting values.

Version 1.5.3 (25 May 2006)

- In function `get.real` a fix was made to accommodate constant pims and a warning is given if the v-c matrix for the betas has non-positive variances.
- In function `make.mark.model`, the argument `initial` can now be a single value which is then assigned as the initial value for all betas. I have found this useful for POPAN models. For some models I have run, the models fail to converge in MARK with the default initial values it uses (I believe it uses `initial=0`). I have had better luck using `initial=1`. By allowing the use of a single value you can use the same generic starting value for each model without figuring out the number of betas in each model. Also note that you can specify another model that has already been run to use as initial values for a new model and it will match parameter values.
- A bad bug was fixed in `cleanup` which was unfortunately deleting files containing "out", "inp", "res" or "vcv" rather than those having these as extensions. This happened without your knowledge if you chose ask=FALSE. Good thing I had a backup. Anyhow, I have now restricted it to files that are named by RMark with markxxxx.inp etc where xxxx is a numeric value. Thus if you assign your own basefile name for output files you'll have to delete them manually. Better safe than sorry.

Version 1.5.2 (18 May 2006)

- Two new functions were added in this version. `convert.inp` converts a MARK encounter history input file to an RMark dataframe. This will be particularly useful for those folks who have already been using MARK. Instead of converting and importing their data with `import.chdata` they can use the `convert.inp` to import their .inp file directly. It can also be used to directly import any of the example .inp files that accompany MARK and the MARK electronic book (`http://www.phidot.org/software/mark/docs/book/`). The second new function is only useful for tutorials and for first time users trying to understand the way RMark works. The function `PIMS` displays the full PIM structure or the simplified PIM structure for a parameter in a model. The user does not directly manipulate PIMS in RMark and they are essentially transparent to the user but for those with MARK experience being able to look at the PIMS may help with the transition.

Version 1.5.1 (11 May 2006)

- Functions `compute.link` and `get.link` were added to compute link values rather than the parameter estimates.
- A function `convert.link.to.real` was added to convert link estimates to real parameter estimates. Previously a similar internal function was used within `compute.real` but to provide more flexibility it was put into a separate function.
- An argument `beta` was added to `get.real` to enable it to be changed in the computation of the real parameters rather than always using the values in `results$beta`.
- A function `TransitionMatrix` was added to create a transition matrix for the Psi values. It is provided for all strata including the `subtract.stratum`. Standard errors and confidence intervals can also be returned.
- `make.mark.model` was modified to include `time.intervals` as an element in the mark object.

Version 1.5.0 (9 May 2006)

- If output file already exists user is given option to create mark model from existing files. Only really useful if a bug occurs (which occurred to me from 1.4.9 changes) and once fixed any models already run can be brought into R by running the same model over and specifying the existing base `filename`. Base `filename` values are no longer prefixed with MRK to enable this change.
- On occasion MARK will complete the analysis but fail to create the v-c matrix and v-c file. The code has been modified to skip over the file if it is missing and output a warning.
- Two new functions have been added to ease handling of marklist objects. `merge.mark` merges an unspecified number of marklist and mark model objects into a new marklist with an optional model.table. `remove.mark` can be used to remove mark models from a marklist. See `dipper` for examples of each function.
- Various changes were made to functions that compute real parameter estimates, their standard errors, confidence intervals and variance-covariance matrix. The functions that were changed include `compute.real`,`find.covariates`,`get.real`,`fill.covariates`. For examples, see help for latter two functions.

Version 1.4.9 (3 May 2006)

- Argument `initial` of `make.mark.model` was not working after model simplification was added in v1.2. This was modified to select initial values from the model based on names of design matrix columns rather than column contents which have different numbers of rows depending on the simplification.
- `extract.mark.output` was fixed to extract the correct -2LnL from the output file in situations in which initial values were specified.

Version 1.4.8 (25 April 2006)

- Argument `silent` was added to `mark` and `mark.wrapper` with a default value of `FALSE`. This overcomes the problem described above in 1.4.7.
- Code was added to `collect.model.names` to prevent it from tripping up when files contain an asterisk which R uses for special names.
- Use of T and F was properly changed to TRUE and FALSE in various functions to prevent errors when T or F are R objects.
- Code for naming files was modified to avoid problems when more than 999 analyses were run in the same directory.
- Bug in setting fixed parameters with argument `fixed=list(index=,value=)` was corrected.
- Argument `remove.intercept` was added to parameter definition to force removal of intercept in designs with nested factor interactions with additional factor variables (e.g., `Psi=list(formula=~sex+stratum:tos`

Version 1.4.7 (10 April 2006)

- An error was fixed in the `Psi` simplification code. Note that with the fix in 1.4.2 to trap errors, a side effect is that non-trapped errors that occur in the R code will now fail without any error messages. If the error occurs in making the model, then the model will not be run, but you will not receive a message that the model failed. I may have to make the error trapping a user-settable option to provide better error tracking.

Version 1.4.6 (7 April 2006)

- Assurance code was added to test that the mlogits were properly assigned. An error message will be given if there has been any unforeseen problem created by the simplification. This eliminates any need for the user to check them as described under 1.4.5 above.

Version 1.4.5 (6 April 2006)

- For multistrata models, the code for creating the mlogit links for Psi was not working properly if there was more than one group. This was fixed in this version.
- Simplification of the PIMS has now been extended to include mlogit parameters. That was not a trivial exercise and while I feel confident it is correct, double check the assignment of mlogit links for complex models, as I have not checked many examples at present. Within a stratum, the corresponding elements for Psi for each of the tostratum (movement from stratum to each of the other strata excluding the `subtract.stratum`) should have the same mlogit(xxx) value such that it can properly compute the value for `subtract.stratum` by subtraction.

Version 1.4.4 (4 April 2006)

- By including the test on model failure, errors that would stop program were not being displayed. This has been fixed in this version.

- An error was fixed in using time-varying covariates when some of the design data had been deleted.

Version 1.4.3 (30 March 2006)

- Problem with pop up window has been fixed. It will no longer appear if the model does not converge but the model will show as having failed.
- An error was fixed in extracting output from the MARK output file when for some circumstances the label for beta parameters included spaces. This now works properly.

Version 1.4.2 (14 March 2006)

- Errors in the FORTRAN code were preventing completion of large batch jobs. Now these errors are caught and models that fail are reported and skipped over. Unfortunately, it does require user intervention to close the popup window. Make sure you select Yes to close the window especially if you use the default `invisible=FALSE` such that the window does not appear. If you select No, you will not able to close the window and R will hang.
- A new list element was added to `parameters` in `make.design.data` for parameters such as `Psi` to set the value of `tostratum` that is computed by subtraction. The default is to compute the probabilitity of remaining in the stratum. The following is an example with strata A to D and setting A to be computed by subtraction for each stratum:
ddl=make.design.data(data.processed,
parameters=list(Psi=list(pim.type="constant",subtract.stratum=c("A","A","A","A")),
p=list(pim.type="constant"),S=list(pim.type="constant")))

Version 1.4.1 (11 March 2006)

- A value "constant" was added for the argument `pim.type`. Note that `pim.type` is only used for triangular PIMS. See `make.design.data`
- Some code changes were made to `make.mark.model` which lessen time to create the MARK input file for large models.
- Function `add.design.data` was modified to accomodate robust design and deletion of design data; this was missed in v1.4 changes.
- `model.name` argument in `mark` and `make.mark.model` was not working. This was fixed.

Version 1.4 (9 March 2006)

- Robust design models added. See `robust` for an example.
- Function `cleanup` was modified so warning messages/errors do not occur if no models/files are found.
- Parameters in the design matrix are now ordered in the same consistent arrangement. In prior versions they were arranged based on their order in the argument call.
- Argument `right` was added to `make.design.data`, `add.design.data` and in `design.parameters` of `make.mark.model` to control whether bins are inclusive on the right (default). The `robust` example uses this argument in a call to `mark`.

Version 1.3 (22 Feb 2006)

- Time varying covariates can now be included in the model formula. See `make.mark.model` for details.

- New model types for Known (Known-fate) and Multistrata (CJS with different strata) were added. See Blackduck and mstrata for examples.
- Specific rows of the design data can now be removed for parameters that should not be estimated. Default fixed values can be assigned. The function make.design.data now accepts an argument remove.unused which can be used to automatically remove unused design data for nested models. It's behavior is also determined by the new argument default.fixed in make.mark.model.
- summary.mark now produces a summary object and print.summary.mark prints the summary object. Changes were made to output when se=T.
- A new function merge.occasion.data was created to add occasion specific covariates to the design data.
- New functions mark.wrapper and create.model.list were created to automate running models from a set of model specifications for each model parameter.
- The argument begin.time in process.data can now be a vector to enable a different beginning time for each group.
- An argument pim.type was added to parameter specification to enable using pims with time structure for data sets with a single release cohort for CJS. See make.design.data
- Model lists created with collect.models are now given the class "marklist" which is used with cleanup and print.marklist (see print.mark).
- The function collect.models now places the model.table at the end of the returned list such that each model number in model.table is now the element number in the returned list. Previously it was 1+ that number.
- Input, output, v-c and residual results files from MARK are now stored in the directory containing the .Rdata workspace. They are numbered consecutively and the field output contains the base filename. The function cleanup was created to delete files that are no longer linked to mark or marklist objects.
- Model averaged estimates and standard errors of real parameters can be obtained with the function model.average.

Version 1.2 (4 Oct 2005)

- By default the PIM structure is simplified to use the fewest number of unique parameters. This reduces the size of the design matrix and should reduce run times.
- The above change was made in some versions still numbered 1.1, but it contained an error that caused the links command for MARK to be constructed incorrectly.
- adjust argument has been added to collect.models to enable control of number of parameters and resulting AIC values.
- model.list in model.table and adjust.chat can now also be a list of models created by collect.models which allows operating on sets of models.

Author(s)

Jeff Laake

wwdo.popan *White-winged dove Jolly-Seber POPAN Analysis Details*

Description

This dataset represents 2 years of capture-mark-recapture data collected on uniquely identifiable leg-banded (size 4) white-winged dove captured in Alice, Texas, USA (Latitude 27.25, Longitude -98.07) between mid-February and mid-September during 2009 and 2010. The package was developed such that others could recreate the analysis developed by Collier, B. A., S. R. Kremer, C. D. Mason, J. Stone, K. W. Calhoun, and M. J. Peterson. 2012. Immigration and recruitment in an urban white-winged dove breeding colony. Journal of Fish and Wildlife Management, In review., and see how the data and results were used to estimate population level recruitment (number juveniles in population over number adults in population).

Format

The format is 2 data frames, one for 2009 and one for 2010. 2009 has 5101 unique captures, 2010 has 3502 unique captures.

Prefix Unique band prefix identifier (usually 0914 or 0984)

Suffix Unique band suffix (numeric value)

E0-E13 0/1 representing whether a dove was captured (1) or not captured (0) during that (E) sampling occasion

Age Age class with AHY=after-hatch year and HY=hatch year

Details

White-winged doves were captured (aged: AHY=after-hatch year, HY=hatch year) continuously in baited walk-in dove traps beginning in Februrary and ending in September in each year (2009 and 2010). During 2009 5,101 white-winged doves were captured (2,894 AHY, 2,207 HY) while in 2010 3,502 white-winged doves were captured (3,106 AHY, 486 HY). We used approximately 2 week date windows to categorized our encounter histories for analysis in MARK http://www.phidot.org/software/mark/ via RMark http://cran.r-project.org/web/packages/RMark/index.html using these dates: 27 Feb; 13 March; 27 March; 10 April; 24 April; 8 May; 22 May; 5 June; 19 June; 3 July; 17 July; 31 July; 14 August-End; giving us 13 encounter occasions.

I wanted to force b0=0 for the first time frame, as none could be there when we started as they had not arrived yet in any real number. If you take the first column out, the numbers get ridiculously screwy for the super population size and the entry parameters, because the initial population has individuals in it, thus the JSPOPAN estimates something like 40 the population pre-trapping, which is biologically impossible.

Initially, because of the parameter structure in a JS-POPAN model and the fact that the initial entry probability is 1 minus the sum of the resultant entry probabilities for subsequent sampling occasions, and because occasionally a couple of doves were captured during the initial time frame, we were getting entry values for the initial time period representing >40 To obtain reasonable results a 'fake' encounter occasion (time -1) with no captures ("0") was appended to beginning of each capture

history to force b0=0. As such, the data for wwdo.09 and wwdo.10 will have 14, not 13 encounter histories.

Important to note, in case you don't read Collier et al. (2012), is the fact that the 2010 dataset is kind of screwy relative to the estimation of the entry parameters for HY wwdo. Basically, what happened was we caught a bunch of AHY birds, but when we captured HY birds, we only caught a few (~400 in 2010) and of those we captured, we rarely, if ever had any recaptures. Without getting into a bunch of speculation on what happened, as we don't really know, we suspect it had something to do with the fact that 2009 was a extreme drought in Texas, as to where 2010 was extremely wet, so mast based food sources (mulberry tree's are everywhere) and such were readily available in the urban environment, as such, lower trapping success. So, when you fit the 2010 dataset using the same candidate model set at the 2009, you get pretty nonsensical answers for the entry b parameters. As such, we did not use a group specific entry model for 2010. If you want more detail, see Collier et al. (2012).

Within the model code below, you can see that we fixed the both the Phi and p parameters for those periods in the analysis when HY birds could not be in the population (e.g., when all birds migrated into the breeding colony and no HY birds had been produced yet). There is probably a little slack in the range, as it is possible that there were some HY white-winged doves in the population in the last period we fixed, but we did not catch any there so we opted to fix it as well.

Note that the R function wwdo.popan is set up in RMark speak, and will run either the wwdo.09 or wwdo.10 datasets as long as you specify one in the lines where I have listed data(wwdo.09) and wwdo=wwdo.09. If you want to see the 2010 analysis, just change those lines to wwdo.10.

References

Collier, B. A., S. R. Kremer, C. D. Mason, J. Stone, K. W. Calhoun, and M. J. Peterson. 2012. Immigration and recruitment in an urban white-winged dove breeding colony. Journal of Fish and Wildlife Management, In review.

Examples

```
data(wwdo.09)
wwdo=wwdo.09
wwdo.popan=function(){
wwdo.proc=process.data(wwdo, model="POPAN", groups="Age")
wwdo.ddl=make.design.data(wwdo.proc)

#Fixing Phi Parameters for sampling periods where HY WWDO were not available in population
hy.phi1=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                        wwdo.ddl$Phi$time==1,]))
hy.phi2=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                        wwdo.ddl$Phi$time==2,]))
hy.phi3=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                        wwdo.ddl$Phi$time==3,]))
hy.phi4=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                        wwdo.ddl$Phi$time==4,]))
hy.phi5=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                        wwdo.ddl$Phi$time==5,]))
hy.phi6=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                        wwdo.ddl$Phi$time==6,]))
```

```
hy.phi7=as.numeric(row.names(wwdo.ddl$Phi[wwdo.ddl$Phi$group=="HY" &
                      wwdo.ddl$Phi$time==7,]))
hy.phi.fix=c(hy.phi1, hy.phi2, hy.phi3, hy.phi4, hy.phi5, hy.phi6, hy.phi7)

#Fixing PENT Parameters for sampling period where HY WWDO were not available in population
hy.pent2=as.numeric(row.names(wwdo.ddl$pent[wwdo.ddl$pent$group=="HY" &
                      wwdo.ddl$pent$time==2,]))
hy.pent3=as.numeric(row.names(wwdo.ddl$pent[wwdo.ddl$pent$group=="HY" &
                      wwdo.ddl$pent$time==3,]))
hy.pent4=as.numeric(row.names(wwdo.ddl$pent[wwdo.ddl$pent$group=="HY" &
                      wwdo.ddl$pent$time==4,]))
hy.pent5=as.numeric(row.names(wwdo.ddl$pent[wwdo.ddl$pent$group=="HY" &
                      wwdo.ddl$pent$time==5,]))
hy.pent6=as.numeric(row.names(wwdo.ddl$pent[wwdo.ddl$pent$group=="HY" &
                      wwdo.ddl$pent$time==6,]))
hy.pent7=as.numeric(row.names(wwdo.ddl$pent[wwdo.ddl$pent$group=="HY" &
                      wwdo.ddl$pent$time==7,]))
hy.pent.fix=c(hy.pent2, hy.pent3, hy.pent4, hy.pent5, hy.pent6, hy.pent7)

#####
#Real Parameter Definitions
#####
#Detection process
p.dot=list(formula=~1)
p.time=list(formula=~time)
p.group=list(formula=~group)
p.g.time=list(formula=~group:time)

#Survival process
Phi.dot.fix=list(formula=~1, fixed=list(index=hy.phi.fix, value=c(0,0,0,0,0,0,0)))
Phi.time.fix=list(formula=~time, fixed=list(index=hy.phi.fix, value=c(0,0,0,0,0,0,0)))
Phi.age.fix=list(formula=~group, fixed=list(index=hy.phi.fix, value=c(0,0,0,0,0,0,0)))
Phi.timeage.fix=list(formula=~time:group,
     fixed=list(index=hy.phi.fix, value=c(0,0,0,0,0,0,0)))

#Entry Process-always time dependent, otherwise makes no sense in my situation
pent.time.fix=list(formula=~time, fixed=list(index=hy.pent.fix, value=c(0,0,0,0,0,0)))

Model.1=mark(wwdo.proc, wwdo.ddl,
   model.parameters=list(Phi=Phi.dot.fix, p=p.dot, pent=pent.time.fix, N=list(formula=~group)),
    invisible=FALSE)
Model.2=mark(wwdo.proc, wwdo.ddl,
  model.parameters=list(Phi=Phi.time.fix, p=p.dot, pent=pent.time.fix, N=list(formula=~group)),
    invisible=FALSE)
Model.3=mark(wwdo.proc, wwdo.ddl,
  model.parameters=list(Phi=Phi.age.fix, p=p.dot, pent=pent.time.fix, N=list(formula=~group)),
     invisible=FALSE)
Model.4=mark(wwdo.proc, wwdo.ddl,
  model.parameters=list(Phi=Phi.timeage.fix, p=p.dot, pent=pent.time.fix, N=list(formula=~group)),
     invisible=FALSE)
Model.5=mark(wwdo.proc, wwdo.ddl,
  model.parameters=list(Phi=Phi.timeage.fix, p=p.time, pent=pent.time.fix, N=list(formula=~group)),
    invisible=FALSE)
```

```
Model.6=mark(wwdo.proc, wwdo.ddl,
   model.parameters=list(Phi=Phi.timeage.fix,p=p.g.time, pent=pent.time.fix,N=list(formula=~group)),
    invisible=FALSE)
collect.models()
}
wwdo.out=wwdo.popan()
wwdo.out
```

Index