Package ‘GeneticsPed’

January 14, 2017

Title Pedigree and genetic relationship functions

Description Classes and methods for handling pedigree data. It also includes functions to calculate genetic relationship measures as relationship and inbreeding coefficients and other utilities. Note that package is not yet stable. Use it with care!

Author Gregor Gorjanc and David A. Henderson <DNADavenator@GMail.Com>, with code contributions by Brian Kinghorn and Andrew Percy (see file COPYING)

Maintainer David Henderson <DNADavenator@GMail.Com>

URL http://rgenetics.org

License LGPL (>= 2.1) | file LICENSE

biocViews Genetics

Version 1.36.0

Depends R (>= 2.4.0), MASS

Imports gdata, genetics

Date 2007-09-20

Suggests RUnit, gtools

NeedsCompilation yes

R topics documented:

check .................................................. 2
extend .................................................. 4
family .................................................. 5
geneContribution .................................. 6
geneFlowT ........................................... 7
generatePedigree ................................. 9
generation .......................................... 10
gpi ..................................................... 11
hwp .................................................... 14
inbreeding .......................................... 15
isFounder ........................................... 17
model.matrix.Pedigree .......................... 18
Mmode ................................................. 19
nIndividual ........................................ 20
check

Description

check performs a series of checks on pedigree object to ensure consistency of data.

Usage

check(x, ...) checkId(x)

Arguments

x pedigree, object to be checked
... arguments to other methods, none for now

Details

checkId performs various checks on individuals and their ascendants. These checks are:

- idClass: all ids must have the same class
- idIsNA: individual can not be NA
- idNotUnique: individual must be unique
- idEqualAscendant: individual can not be equal to its ascendant
- ascendantEqualAscendant: ascendant can not be equal to another ascendant
- ascendantInAscendant: ascendant can not appear again as ascendant of other sex i.e. father can not be a mother to someone else
- unusedLevels: in case factors are used for id presentation, there might be unused levels for some ids - some functions rely on number of levels and a check is provided for this

checkAttributes is intended primarily for internal use and performs a series of checks on attribute values needed in various functions. It causes stop with error messages for all given attribute checks.

Value

List of more or less self-explanatory errors and "pointers" to these errors for ease of further work i.e. removing errors.

Author(s)

Gregor Gorjanc
check

See Also

Pedigree

Examples

## EXAMPLES BELOW ARE ONLY FOR TESTING PURPOSES AND ARE NOT INTENDED
## FOR USERS, BUT IT CAN NOT DO ANY HARM.

```r
## --- checkAttributes ---
tmp <- generatePedigree(5)
attr(tmp, "sorted") <- FALSE
attr(tmp, "coded") <- FALSE
GeneticsPed:::checkAttributes(tmp)
try(GeneticsPed:::checkAttributes(tmp, sorted=TRUE, coded=TRUE))

## --- idClass ---
tmp <- generatePedigree(5)
tmp$id <- factor(tmp$id)
class(tmp$id)
class(tmp$father)
try(GeneticsPed:::idClass(tmp))

## --- idIsNA ---
tmp <- generatePedigree(2)
tmp[1, 1] <- NA
GeneticsPed:::idIsNA(tmp)

## --- idNotUnique ---
tmp <- generatePedigree(2)
tmp[2, 1] <- 1
GeneticsPed:::idNotUnique(tmp)

## --- idEqualAscendant ---
tmp <- generatePedigree(2)
tmp[3, 2] <- tmp[3, 1]
GeneticsPed:::idEqualAscendant(tmp)

## --- ascendantEqualAscendant ---
tmp <- generatePedigree(2)
tmp[3, 2] <- tmp[3, 3]
GeneticsPed:::ascendantEqualAscendant(tmp)

## --- ascendantInAscendant ---
tmp <- generatePedigree(2)
tmp[3, 2] <- tmp[5, 3]
GeneticsPed:::ascendantInAscendant(tmp)

## --- ascendantInAscendant ---
tmp <- data.frame(id=c("A", "B", "C", "D"),
                 father1=c("E", NA, "F", "H"),
                 father2=c("F", "E", "E", "I"),
                 mother=c("G", NA, "H", "E"))
tmp <- Pedigree(tmp, ascendant=c("father1", "father2", "mother"),
                ascendantSex=c(1, 1, 2),
                ascendantLevel=c(1, 1, 1))
GeneticsPed:::ascendantInAscendant(tmp)
```
## --- unusedLevels ---

tmp <- generatePedigree(2, colClass="factor")
tmp[3:4, 2] <- NA
GeneticsPed:::unusedLevels(tmp)

---

**extend**

**Extend pedigree**

**Description**

extend finds ascendants, which do not appear as individuals in pedigree and assigns them as individuals with unknown ascendants in extended pedigree.

**Usage**

```r
extend(x, ascendant=NULL, col=NULL, top=TRUE)
```

**Arguments**

- `x`: pedigree object
- `ascendant`: character, column names of ascendant(s), see details
- `col`: character, column name(s) of attribute(s), see details
- `top`: logical, add ascendants as individuals on the top or bottom of the pedigree

**Details**

Argument `ascendant` can be used to define, which ascendants will be extended. If `ascendant=NULL`, which is the default, all ascendant columns in the pedigree are used. The same approach is used with other pedigree attributes such as sex, generation, etc. with argument `col`. Use `col=NULL`, if none of the pedigree attributes should be extended.

Sex of “new” individuals is inferred from attribute `ascendantSex` as used in `Pedigree` function. Generation of “new” individuals is inferred as minimal (`generationOrder="increasing"`) or maximal (`generationOrder="decreasing"`) generation value in descendants - 1. See `Pedigree` on this issue. Family values are extended with means of `family`.

**Value**

Extended pedigree, where all ascendants also appear as individuals with unknown ascendants and infered other attributes such as sex, generation, etc. if this attributes are in the pedigree.

**Author(s)**

Gregor Gorjanc

**See Also**

`Pedigree`, `family`, `geneticGroups`
Examples

# --- Toy example ---
ped <- generatePedigree(nId=5, nGeneration=4, nFather=1, nMother=2)
ped <- ped[10:20,]
ped[5, "father"] <- NA # to test robustness of extend on NA
extend(ped)
extend(ped, top=FALSE)

## Extend only ascendant and their generation
extend(ped, col="generation")
extend(ped, col=c("generation", "sex"))

# --- Bigger example ---
ped <- generatePedigree(nId=1000, nGeneration=10, nFather=100,
                        nMother=500)
nrow(ped)
# Now keep some random individuals
ped <- ped[unique(sort(round(runif(n=nrow(ped)/2, min=1,
                              max=nrow(ped)))), )]
nrow(ped)
nrow(extend(ped))

family

Find families (lines) in the pedigree

Description

family classifies individuals in the pedigree to distinct families or lines. Two individuals are members of one family if they have at least one common ascendant. family<- provides mean to properly add family information into the pedigree.

Usage

family(x)
family(x, col=NULL) <- value

Arguments

x 
pedigree object

col 
character, column name in x for family

value 
family values for individuals in the pedigree

Details

col provides a mean to name or possibly also rename family column with user specified value, say "familia" in Spanish. When col=NULL, which is default, “family” is used.

Value

A vector of family values (integers)
## Examples

```r
# Two families examples
ped <- data.frame(id=c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11),
                  father=c(0, 0, 0, 0, 0, 0, 5, 1, 3, 8, 7),
                  mother=c(0, 0, 0, 0, 0, 0, 6, 2, 4, 9, 10),
                  generation=c(1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4))
ped <- Pedigree(ped, unknown=0, generation="generation")
family(ped)

# After break we get two families
ped1 <- removeIndividual(ped, individual=11)
family(ped1)

# Subsetting can also be used
family(ped[1:10,])
family(ped[7:10,])

# Pedigree need not be sorted in advance
ped2 <- ped[sample(1:10), ]
family(ped2)

# Assign family values to pedigree
family(ped) <- family(ped)
ped
family(ped, col="familia") <- family(ped)
ped
```

---

geneContribution  

**Gene contribution or proportion of genes in pedigree by individual**

### Description

geneContribution calculates gene contribution as proportion of genes in pedigree by individual with higher number of descendants will have higher values.

### Usage

geneContribution(x, relative=TRUE)

### Arguments

- `x`  
  pedigree

- `relative`  
  logical, should results be presented relative to number of individuals in the pedigree
Value

Gene contribution values i.e. higher the values higher the contribution of genes by particular individual in the pedigree. When `relative=FALSE`, values represent number of individuals (in conceptually additive manner i.e. $0.5 + 0.75 = 1.25$ individual) in the pedigree that carry genes of a particular individual. With `relative=TRUE`, values represent the same result as ratios to all individuals in the pedigree. Value 0 indicates that individual did not pass its genes to next generations.

Author(s)

Gregor Gorjanc

See Also

Pedigree

Examples

```r
ped <- generatePedigree(nId=5, nGeneration=4, nFather=1, nMother=2)
geneContribution(ped)
geneContribution(ped, relative=FALSE)
## geneContribution(ped[5:15, ]) ## needs [ method

## More than one father example
ped <- data.frame(
  id=c(1, 2, 3, 4, 5, 6, 7),
  father1=c(0, 0, 0, 2, 1, 1, 2),
  father2=c(0, 0, 0, 0, 0, 2, 0),
  mother=c(0, 0, 0, 3, 3, 3, 3),
  generat=c(1, 1, 1, 2, 2, 2, 2))
ped <- Pedigree(ped, ascendant=c("father1", "father2", "mother"),
  ascendantSex=c(1, 1, 2), ascendantLevel=c(1, 1, 1),
  unknown=0, generation="generat")
geneContribution(ped)
```

geneFlowT

Gene and gamete flow matrices

Description

geneFlowT and geneFlowTinv creates gene flow matrix (T) and its inverse (Tinv), while gameteFlowM creates gamete flow matrix (M). mendelianSamplingD creates a mendelian sampling covariance matrix (D).

Usage

```r
geneFlowT(x, sort=TRUE, names=TRUE, ...)
geneFlowTinv(x, sort=TRUE, names=TRUE, ...)
gameteFlowM(x, sort=TRUE, names=TRUE, ...)
mendelianSamplingD(x, matrix=TRUE, names=TRUE, ...)```
Arguments

- **x**: Pedigree
- **sort**: logical, for the computation the pedigree needs to be sorted, but results are sorted back to original sorting (sort=TRUE) or not (sort=FALSE)
- **names**: logical, should returned matrix have row/colnames; this can be used to get leaner matrix
- **matrix**: logical, should returned value be a diagonal matrix or a vector
- ... arguments for other methods

Details

geneFlowT returns a matrix with coefficients that show the flow of genes from one generation to the next one etc. geneFlowTinv is simply the inverse of geneFlowT, but calculated as \( I - M \), where \( M \) is gamete flow matrix with coefficients that represent parent gamete contribution to their offspring. mendelianSamplingD is another matrix \((D)\) for construction of relationship additive matrix via decomposition i.e. \( A = TDT' \) (Henderson, 1976). Mrode (2005) has a very nice introduction to these concepts.

Take care with sort=FALSE, names=FALSE. It is your own responsibility to assure proper handling in this case.

Value

Matrices of \( n \times n \) dimension, with coefficients as described in the details, where \( n \) is number of subjects in \( x \)

Author(s)

Gregor Gorjanc

References


See Also

`Pedigree`, `relationshipAdditive`, `kinship` and `inbreeding`

Examples

```r
if(require(gdata))
data(Mrode2.1)
Mrode2.1$dtB <- as.Date(Mrode2.1$dtB)
x2.1 <- Pedigree(x=Mrode2.1, subject="sub", ascendant=c("fat", "mot"),
                      ascendantSex=c("M", "F"), family="fam", sex="sex",
                      generation="gen", dtBirth="dtB")

fractions(geneFlowT(x2.1))
fractions(geneFlowTinv(x2.1))
fractions(gameteFlowM(x2.1))
mendelianSamplingD(x2.1)
```
### Description

`generatePedigree` creates nonoverlapping pedigree example, which can be used for demos and code testing.

### Usage

```r
generatePedigree(nId, nGeneration=3, nFather=round(nId/3),
                 nMother=nId - nFather, start=1, generationOrder="increasing",
                 colClass="integer")
```

### Arguments

- **nId**: integer, number of individuals per generation, at least 2
- **nGeneration**: integer, number of generations
- **nFather**: integer, number of fathers per generation
- **nMother**: integer, number of mothers per generation
- **start**: first generation value
- **generationOrder**: character, generation value is "increasing" or "decreasing" through generations
- **colClass**: character, class for columns: "integer" or "factor"

### Value

An extended, sorted and possibly coded pedigree object with following columns: id, father, mother, generation and sex.

### Author(s)

Marcos Rico Gutierrez (author of MATLAB code) and Gregor Gorjanc (R implementation)

### References


### See Also

Pedigree

### Examples

```r
generatePedigree(5)
generatePedigree(nId=5, nGeneration=4, nFather=1, nMother=2)
generatePedigree(nId=5, nGeneration=4, nFather=1, nMother=2,
                 start=0, generationOrder="decreasing")
generatePedigree(nId=100, nGeneration=20, nFather=10, nMother=50)
```
**generation**  

*Calculate generation value*

**Description**

`generation` calculates generation value of individuals in given pedigree. `generation<-` provides a way to properly add generation information into the pedigree.

**Usage**

```r
 generation(x, start=1, generationOrder=NULL)  
generation(x, generationOrder=NULL, col=NULL) <- value
```

**Arguments**

- `x` pedigree object
- `start` first generation value
- `generationOrder` character, should be generation values "increasing" or "decreasing" through generations, see details
- `col` character, column name in `x` for generation
- `value` generation values for individuals in the pedigree

**Details**

Generation value for founders is set to value `start`, which is by default 1, while other individuals get it according to:

\[ G_s = \text{max}(G_{1a} + G_{2a} + ...G_{na}) + 1 \]

where `G` represents generation value for `s` - individual, `a` - ascendant e.g. father and mother, where `n=2`. `N` might be higher if there are multiple ascendants i.e. this function can also handle pedigrees with higher order ascendants e.g. grandfather.

`generationOrder` can be used to define "increasing" or "decreasing" order of generation values. If this argument is `NULL`, which is default, then this information is taken from the pedigree - see `Pedigree` for more on this issue.

`col` provides a way to name or possibly also rename generation column with user specified value, say "generazione" in Italian. When `col=NULL`, which is default, "generation" is used.

**Value**

A vector of generation values (integers)

**Author(s)**

Gregor Gorjanc

**See Also**

`Pedigree`
Examples

```
# Nonoverlapping pedigree
ped <- generatePedigree(nId=5, nGeneration=4, nFather=1, nMother=2)
ped$generation1 <- generation(ped)
ped

# Overlapping Pedigree
ped <- data.frame( id=c(1, 2, 3, 4, 5, 6, 7),
                  father=c(0, 0, 2, 2, 2, 4, 4),
                  mother=c(0, 0, 1, 0, 3, 3, 5),
                  dtBirth=c(2, 1, 3, 4, 5, 6, 7))
ped <- Pedigree(ped, unknown=0, dtBirth="dtBirth")
generation(ped) <- generation(ped)

# Overlapping pedigree + one individual (4) comes late in pedigree and
# has no ascendants
ped <- data.frame( id=c(1, 2, 3, 4, 5, 6, 7),
                  father=c(0, 0, 2, 0, 2, 4, 4),
                  mother=c(0, 0, 1, 0, 3, 3, 5),
                  dtBirth=c(2, 1, 3, 2, 5, 6, 7))
ped <- Pedigree(ped, unknown=0, dtBirth="dtBirth")
generation(ped)
generation(ped, generationOrder="decreasing",
           col="generazione") <- generation(ped, generationOrder="decreasing")
```

---

**gpi**

*Genotype probability index*

**Description**

gpi calculates Genotype Probability Index (GPI), which indicates the information content of genotype probabilities derived from segregation analysis.

**Usage**

gpi(gp, hwp)

**Arguments**

- `gp` numeric vector or matrix, individual genotype probabilities
- `hwp` numeric vector or matrix, Hard-Weinberg genotype probabilities

**Details**

Genotype Probability Index (GPI; Kinghorn, 1997; Percy and Kinghorn, 2005) indicates information that is contained in multi-allele genotype probabilities for diploids derived from segregation analysis, say Thallman et. al (2001a, 2001b). GPI can be used as one of the criteria to help identify which ungenotyped individuals or loci should be genotyped in order to maximise the benefit of genotyping in the population (e.g. Kinghorn, 1999).

gp and hwp arguments accept genotype probabilities for multi-allele loci. If there are two alleles (1 and 2), you should pass vector of probabilities for genotypes (11 and 12) i.e. one value for heterozygotes (12 and 21) and always skipping last homozygote. With three alleles this vector
should hold probabilities for genotypes (11, 12, 13, 22, 23) as also shown bellow and in examples. 
\texttt{hwp} and \texttt{gpLong2Wide} functions can be used to ease the setup for \texttt{gp} and \texttt{hwp} arguments.

2 alleles: 1 and 2  
11 12  
--> no. dimensions = 2

3 alleles: 1, 2, and 3  
11 12 13  
22 23  
--> no. dimensions = 5

...  

5 alleles: 1, 2, 3, 4, and 5  
11 12 13 14 15  
22 23 24 25  
33 34 35  
44 45  
--> no. dimensions = 14

In general, number of dimensions ($k$) for $n$ alleles is equal to:

$$k = (n \times (n + 1)/2) - 1.$$  

If you have genotype probabilities for more than one individual, you can pass them to \texttt{gp} in a matrix form, where each row represents genotype probabilities of an individual. In case of passing matrix to \texttt{gp}, \texttt{hwp} can still accept a vector of Hardy-Weinberg genotype probabilities, which will be used for all individuals due to recycling. If \texttt{hwp} also gets a matrix, then it must be of the same dimension as that one passed to \texttt{gp}.

**Value**

Vector of $N$ genotype probability indices, where $N$ is number of individuals

**Author(s)**

Gregor Gorjanc R code, documentation, wrapping into a package; Andrew Percy and Brian P. Kinghorn Fortran code

**References**


See Also

`hwp` and `gpLong2Wide`

Examples

```r
## --- Example 1 from Percy and Kinghorn (2005) ---
## No. alleles: 2
## No. individuals: 1
## Individual genotype probabilities:
## Pr(11, 12, 22) = (.1, .5, .4)
##
## Hardy-Weinberg probabilities:
## Pr(1, 2) = (.75, .25)
##
## gp <- c(.1, .5)
## hwp <- c(.5625, .3750)
## gpi(gp=gp, hwp=hwp)

## --- Example 1 from Percy and Kinghorn (2005) extended ---
## No. alleles: 2
## No. individuals: 2
## Individual genotype probabilities:
## Pr_1(11, 12, 22) = (.1, .5, .4)
## Pr_2(11, 12, 22) = (.2, .5, .3)
##
## gp <- matrix(c(.1, .5, .2, .5), nrow=2, ncol=2, byrow=TRUE)
## gpi(gp=gp, hwp=hwp)

## --- Example 2 from Percy and Kinghorn (2005) ---
## No. alleles: 3
## No. individuals: 1
## Individual genotype probabilities:
## Pr(11, 12, 13, 22, 23, 33) = (.1, .5, .0, .4, .0, .0)
##
## Hardy-Weinberg probabilities:
## Pr(1, 2, 3) = (.75, .25, .0)
##
## (gp <- matrix(c(.1, .5, .2, .5), nrow=2, ncol=2, byrow=TRUE))
## gpi(gp=gp, hwp=hwp)
```

```r
## --- Example 2 from Percy and Kinghorn (2005) extended ---
## No. alleles: 3
## No. individuals: 2
## Individual genotype probabilities:
## Pr(11, 12, 13, 22, 23, 33) = (.1, .5, .0, .4, .0, .0)
##
## Hardy-Weinberg probabilities:
## Pr(1, 2, 3) = (.75, .25, .0)
##
## (gp <- matrix(c(.1, .5, .2, .5), nrow=2, ncol=2, byrow=TRUE))
## gpi(gp=gp, hwp=hwp)
```
gp <- c(.1, .5, .0, .4, .0)
hwp <- c(.5625, .375, .0, .0625, .0)
gpi(gp=gp, hwp=hwp)

### --- Example 3 from Percy and Kinghorn (2005) ---
### No. alleles: 5
### No. individuals: 1
### Hardy-Weinberg probabilities:
### \( P_r(1, 2, 3, 4, 5) = (.2, .2, .2, .2, .2) \)
### \( P_r(11, 12, 13, \ldots) = (P_r(1)^2, 2P_r(1)+P_r(2), 2P_r(1)P_r(3), \ldots) \)
### Individual genotype probabilities:
### \( P_r(11, 12, 13, \ldots) = \frac{gp}{2} \)
### \( P_r(12) = P_r(12) + .5 \)

(hwp <- rep(.2, times=5) %*% t(rep(.2, times=5)))
hwp <- c(hwp[upper.tri(hwp, diag=TRUE)])
(hwp <- hwp[1:(length(hwp) - 1)])
gp <- hwp / 2
gp

gpi(gp=gp, hwp=hwp)

### --- Simulate gp for n alleles and i individuals ---
n <- 3
i <- 10

kAll <- (n*(n+1)/2) # without -1 here!
k <- kAll - 1
if(require("gtools")) {
  gp <- rdirichlet(n=i, alpha=rep(x=1, times=kAll))[1:k]
  hwp <- as.vector(rdirichlet(n=1, alpha=rep(x=1, times=kAll))[1:k]
  gpi(gp=gp, hwp=hwp)
}

---

### Utility functions for gpi()

**Description**

- gpLong2Wide changes data.frame with genotype probabilities in long form (one genotype per row) to wide form (one individual per row) for use in gpi.
- hwp calculates genotype probabilities according to Hardy-Weinberg law for use in gpi.

**Usage**

- gpLong2Wide(x, id, genotype, prob, trim=TRUE)
- hwp(x, trim=TRUE)
### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>data.frame for gpLong2Wide, genotype for hwp</td>
</tr>
<tr>
<td><code>id</code></td>
<td>character, column name in x holding individual identifications</td>
</tr>
<tr>
<td><code>genotype</code></td>
<td>character, column name in x holding genotypes</td>
</tr>
<tr>
<td><code>prob</code></td>
<td>character, column name in x holding genotype probabilities</td>
</tr>
<tr>
<td><code>trim</code></td>
<td>logical, remove last column (for gpLong2Wide) or value (for hwp) of a result</td>
</tr>
</tbody>
</table>

### Details

Hardy-Weinberg probabilities for a gene with two alleles A and B, with probabilities $Pr(A)$ and $Pr(B)$ are:

- $Pr(AA) = Pr(A)^2$
- $Pr(AB) = 2 \times Pr(A) \times Pr(A)$
- $Pr(BB) = Pr(B)^2$

### Value

- `gpLong2Wide` returns a matrix with number of rows equal to number of individuals and number of columns equal to number of possible genotypes.
- `hwp` returns a vector with Hardy-Weinberg genotype probabilities.

### Author(s)

Gregor Gorjanc

### See Also

`gpi`, `genotype`, `expectedGenotypes`

### Examples

```r
if(require(genetics)) {
  gen <- genotype(c("A/A", "A/B"))
  hwp(x=gen)
  hwp(x=gen, trim=FALSE)
}
```

### Description

`inbreeding` calculates inbreeding coefficients of individuals in the pedigree

### Usage

```r
inbreeding(x, method="meuwissen", sort=TRUE, names=TRUE, ...)```
Arguments

x pedigree object

method character, method of calculation "tabular", "meuwissen" or "sargolzaei", see details

sort logical, for the computation the pedigree needs to be sorted, but results are sorted back to original sorting (sort=TRUE) or not (sort=FALSE)

names logical, should returned vector have names; this can be used to get leaner returned object

Details

Coefficient of inbreeding \( F \) represents probability that two alleles on a loci are identical by descent (Wright, 1922; Falconer and Mackay, 1996). Wright (1922) showed how \( F \) can be calculated but his method of paths is not easy to wrap in a program. Calculation of \( F \) can also be performed using tabular method for setting the additive relationship matrix (Henderson, 1976), where \( F_i = A_{ii} - 1 \). Meuwissen and Luo (1992) and VanRaden (1992) developed faster algorithms for \( F \) calculation. Wiggans et al. (1995) additionally explains method in VanRaden (1992). Sargolzaei et al. (2005) presented yet another fast method.

Take care with sort=FALSE, names=FALSE. It is your own responsibility to assure proper handling in this case.

Value

A vector of length \( n \) with inbreeding coefficients, where \( n \) is number of subjects in \( x \)

Author(s)

Gregor Gorjanc and Dave A. Henderson

References

isFounder

See Also

Pedigree, relationshipAdditive, kinship and geneFlowT

Examples

data(Mrode2.1)
Mrode2.1$dtB <- as.Date(Mrode2.1$dtB)
x2.1 <- Pedigree(x=Mrode2.1, subject="sub", ascendant=c("fat", "mot"),
                  ascendantSex=c("M", "F"), family="fam", sex="sex",
                  generation="gen", dtBirth="dtB")
fractions(inbreeding(x=x2.1))

## Compare the speed
ped <- generatePedigree(nId=25)
system.time(inbreeding(x=ped))
# system.time(inbreeding(x=ped, method="sargolzaei")) # not yet implemented
system.time(inbreeding(x=ped, method="tabular"))

isFounder Founder and non-founder individuals in the pedigree

Description

isFounder classifies individuals in the pedigree as founders (base) or non-founders (non-base individuals).

Usage

isFounder(x, col=attr(x, ".ascendant"))

Arguments

x pedigree object
col character, which columns should be checked, see examples

Details

By definition founders do not have any known ascendants, while the opposite is the case for non-founders i.e. they have at least one known ascendant.

FIXME: any relation with founderGeneSet in GeneticsBase

Value

Boolean vector.

Author(s)

Gregor Gorjanc

See Also

Pedigree
Examples

```r
ped <- generatePedigree(nId=5)
isFounder(ped)
## Based only on fathers
isFounder(ped, col=c("father"))
## Works also only on a part of a pedigree
isFounder(ped[1:5, ])
```

model.matrix.Pedigree  Model matrix for individuals with and without records

Description

`model.matrix` for pedigree creates design matrix (Z) for individuals with and without records. Used mainly for educational purposes.

Usage

```r
## S3 method for class 'Pedigree'
model.matrix(object, y, id, left=TRUE, names=TRUE, ...
```

Arguments

- `object` Pedigree
- `names` logical, should returned matrix have row/colnames; this can be used to get leaner matrix
- `y` numeric, vector of (phenotypic) records
- `id` vector of subjects for y
- `left` logical, bind columns of individuals without records to left (left=TRUE) or right (left=FALSE) side of Z
- `...` arguments passed to `model.matrix`

Value

A model matrix of \( n \times q \) dimension, where \( n \) is number of records in \( y \) and \( q \) is number of subjects in the pedigree

Author(s)

Gregor Gorjanc

See Also

`Pedigree`, `relationshipAdditive`, `inverseAdditive` and `model.matrix`

Examples

```r
data(Mrode3.1)
(x <- Pedigree(x=Mrode3.1, subject="calf", ascendant=c("sire", "dam"),
            ascendantSex=c("Male", "Female"), sex="sex"))
model.matrix(object=x, y=x$pwg, id=x$calf)
```
Mrode

Description

Various pedigree and data examples

Usage

data(Falconer5.1)
data(Mrode2.1)
data(Mrode3.1)

Format

Falconer5.1 is a rather complex (inbreed) pedigree example from book by Falconer and Mackay (1996) - page 84 with 18 individuals and following columns:

sub individual
fat father
mot mother

Mrode2.1 is an extended pedigree example from book by Mrode (2005) - page 27 with 6 individuals and following columns:

sub individual
fat father
mot mother
fam family
sex sex
gen generation
dtB date of birth

Mrode3.1 is a pedigree and data example from book by Mrode (2005) - page 43: it shows a beef breeding scenario with 8 individuals (animals), where 5 of them have phenotypic records (pre-weaning gain) and 3 three of them are without records and link others through the pedigree:

calf calf
sex sex of a calf
sire father of a calf
dam mother of a calf
pwg pre-weaning gain of a calf in kg

References


Examples

data(Falconer5.1)
Pedigree(x=Falconer5.1, subject="sub", ascendant=c("fat", "mot"))

data(Mrode2.1)
Mrode2.1$dtB <- as.Date(Mrode2.1$dtB)
Pedigree(x=Mrode2.1, subject="sub", ascendant=c("fat", "mot"),
  ascendantSex=c("M", "F"), family="fam", sex="sex",
  generation="gen", dtBirth="dtB")

data(Mrode3.1)
Pedigree(x=Mrode3.1, subject="calf", ascendant=c("sire", "dam"),
  ascendantSex=c("Male", "Female"), sex="sex")

Description

nIndividual returns number of individuals (individuals and/or ascendants) in a pedigree object.

Usage

nIndividual(x, col=NULL, extend=TRUE, drop=TRUE)

Arguments

x  pedigree
col character, which id column should be the source: "id" (default) or particular
    ascendant i.e. "father" and "mother"
extend logical, extend pedigree
drop logical, drop unused levels in case factors are used

Details

FIXME - this will change a lot!!!! There is always one additional level in levels in case factors are
used to represent individuals in a pedigree as described in Pedigree. However, nlevels.Pedigree
prints out the number of levels actually used to represent individuals i.e. level unknown is not
included into the result.

Author(s)

Gregor Gorjanc

See Also

summary.Pedigree, extend
Examples

# Default example
ped <- generatePedigree(5)
nIndividual(ped)

# Other id columns
nIndividual(ped, col="father")
nIndividual(ped, col="mother")

# Remove individuals with unknown fathers - FIXME
# ped <- ped[!is.na(ped, col="father"), ]
# nIndividual(ped)
# nIndividual(ped, extend=FALSE)

Description

Pedigree function creates a pedigree object

Usage

Pedigree(x, subject="id", ascendant=c("father", "mother"),
ascendantSex=c(1, 2), ascendantLevel=c(1, 1), unknown=NA, sex=NA,
dtBirth=NA, generation=NA, family=NA, generationOrder="increasing",
check=TRUE, sort=FALSE, extend=FALSE, drop=TRUE, codes=FALSE)

Arguments

x data.frame or matrix
subject character, column name in x for subjects
ascendant character, column name(s) in x for ascendants
family character, column name in x for family
ascendantSex integer or character, sex of ascendant(s); see details
ascendantLevel integer, generation level of ascendant(s); see details
unknown vector or list, unknown representation of identification and other data in the pedigree; see details
sex character, column name in x for sex
dtBirth character, column name in x for date of birth
generation character, column name in x for generation
generationOrder character, generation value is "increasing" or "decreasing" through generations; see details
check logical, check for common errors
sort logical, sort pedigree
extend logical, extend pedigree
drop logical, drop unused levels if factors are used
codes logical, code individuals into integers
Pedigree can be one source of information on genetic relationship between relatives. Take for example the following pedigree:

```
paternal  paternal  maternal  maternal
grandfather grandmother grandfather grandmother

------------- -------------

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>

father  mother

-------------------------

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

subject
```

This information can be stored in a data.frame as

```
mother  maternal grandfather  maternal grandmother
father  paternal grandfather  paternal grandmother
subject  father  mother
```

There is considerable variability in terminology between as well as within various fields of genetics. We use the following terms throughout the help and code: individual (any individual in a pedigree), subject (individual whose pedigree is given i.e. individuals in the first column in upper data.frame), ascendant and descendant. Additionally, family, sex, dtBirth and generation are used for additional data in the pedigree. Their meaning should be clear. For these, argument col is usually used in function calls.

**family**: TODO

**ascendantSex** can be used to define sex of ascendant(s); for example c("Male", "Female") or c(“M”, "F") or even c(1, 2) for father and mother or c(2, 1, 1) for mother and two possible fathers or c(1, 1) for father and maternal father etc. This data is needed only for the structure of the class and defaults should be ok for the majority. But you need to make sure that data defined here must be in accordance with values in sex column.

**ascendantLevel** can be used to define generation level of ascendant(s) in relation to a subject; for example c(1, 1) for father and mother or c(1, 1, 1) for mother and two possible fathers or c(1, 2) for father and maternal father etc. This data is needed only for the structure of the class and defaults should be ok for the majority.

There is no need for as.integer TODO in arguments ascendantLevel as this is done internally.

**unknown**: TODO

**Sex**: TODO

**Date of birth**: TODO

**generationOrder** defines in which order are generation values: "increasing" if values increase from ascendants to descendants and "decreasing" if values decrease from ascendants to descendants. **check**, **sort**, **extend**, and **codes** are actions on the pedigree and have their own help pages.

Individuals can be stored as either integer, numeric or factor TODO. In any case all id columns must have the same class and this is automatically checked. Argument drop can be used to drop unused levels, if factors are used.
Object of Pedigree class is a data.frame with columns that can be divided into core columns (subject, ascendant(s), sex, dtBirth and generation) and possibly other columns such as data on phenotype and genotype and other subject attributes, for example factors and covariates.

Additionally, the following attributes are set on pedigree:

- .subject: column name of subject identification in pedigree
- .ascendant: column name(s) of ascendant(s) identification in pedigree
- .family: column name of family identification in pedigree
- .ascendantSex: sex of ascendant(s)
- .ascendantLevel: generation level of ascendant(s)
- .sex: column name of subject’s sex
- .dtBirth: column name of subject’s date of birth
- .generation: column name of subject’s generation
- .generationOrder: generation value is "increasing" or "decreasing" through generations
- .colClass: storage class for id columns: "integer", "numeric" or "factor"
- .checked: pedigree checked for common errors
- .sorted: pedigree sorted; by
- .extended: pedigree extended
- .coded: pedigree coded
- .unknown: unknown representation for individual identification and other data in the pedigree; names of the list are c(".id", ".family", ".sex", ".dtBirth", ".generation")

**Value**

Pedigree object as described in the details

**Author(s)**

Gregor Gorjanc

**See Also**

`check`, `sort`, and `extend` provide help on pedigree utility functions.

**Examples**

data(Mrode2.1)
Mrode2.1$dtB <- as.Date(Mrode2.1$dtB)
x2.1 <- Pedigree(x=Mrode2.1, subject="sub", ascendant=c("fat", "mot"),
ascendantSex=c("M", "F"), family="fam", sex="sex",
generation="gen", dtBirth="dtB")
if (FALSE) {
    ## How to handle different pedigree types
    ## * multiple parents
    ped2 <- ped
    ped2$father1 <- ped$father
    ped2$father2 <- ped$father
    ped2$father <- NULL
    ped2 <- as.data.frame(ped2)
    str(Pedigree(ped2, ascendant=c("father1", "father2", "mother"),
                  ascendantSex=c(1, 1, 2), ascendantLevel=c(1, 1, 1)))

    ## * different level of parents
    ped3 <- as.data.frame(ped)
    ped3$m.grandfather <- ped3$mother
    ped3$mother <- NULL
    str(Pedigree(ped3, ascendant=c("father", "m.grandfather"),
                 ascendantSex=c(1, 1), ascendantLevel=c(1, 2)))
}

---

**prune**

Prune pedigree

**Description**

prune removes noninformative individuals from a pedigree. This process is usually called trimming or pruning. Individuals are removed if they do not provide any ancestral ties between other individuals. It is possible to add some additional criteria. See details.

**Usage**

```r
prune(x, id, father, mother, unknown=NA, testAdd=NULL, verbose=FALSE)
```

**Arguments**

- `x`: data.frame, pedigree data
- `id`: character, individuals’s identification column name
- `father`: character, father’s identification column name
- `mother`: character, mother’s identification column name
- `unknown`: value(s) used for representing unknown parent in x
- `testAdd`: logical, additional criteria; see details
- `verbose`: logical, print some more info
Details

NOTE: this function does not yet work with Pedigree class.

There are always some individuals in the pedigree that jut out. Usually this are older individuals without known ancestors, founders. If such individuals have only one (first) descendant and no phenotype/genotype data, then they do not give us any additional information and can be safely removed from the pedigree. This process resembles cutting/pruning the branches of a tree.

By default `prune` iteratively removes individuals from the pedigree (from top to bottom) if:

- they are founders, have both ancestors i.e. father and mother unknown and
- have only one or no (first) descendants i.e. children

If there is a need to take into account availability of say phenotype/genotype data or any other information, argument `testAdd` can be used. Value of this argument must be logical and with length equal to number of rows in the pedigree. The easiest way to achieve this is to `merge` any data to the pedigree and then to perform a test, which will return logical values. Note that value of `TRUE` in `testAdd` means to remove an individual - this function is removing individuals! To keep an individual without known parents and one or no children, value of `testAdd` must be `FALSE` for that particular individual. Take a look at the examples.

There are various conventions on representing unknown/missing ancestors, say 0. R’s default is to use `NA`. If other values than `NA` are present, argument `unknown` can be used to convert unknown/missing values to `NA`.

It is assumed that pedigree is in extended form i.e. that each father and mother has each own record as an individual. Otherwise error is returned with information on which parents do not appear as individuals.

`prune` does not only remove lines for pruned individuals but also removes them from `father` and `mother` columns.

Pruning is done from top to bottom of the pedigree i.e. from oldest individuals towards younger ones. Take for example the following part of the pedigree in example section:

```
0  7
 | |
------
| 10 8
 | |
------
| 9
```

Individual 7 is not removed since it has two (first) descendants i.e. 8 and 5 (not shown here). Consecutively, individuals 8 and 9 are also not removed from the pedigree. Individual 10 is removed, since it has only one descendant. Why should individuals 8 and 9 and therefore also 7 stay in the pedigree? Current behaviour is reasonable if pedigree is built in such a way that first individuals with some phenotype or genotype data are gathered and then their pedigree is being built. Say, individual 9 has phenotype/genotype data and its pedigree is build and there is therefore no need to remove such an individual. However, if pedigree is not built in such a way, then `prunPedigree` function can not prune all noninformative individuals. Argument `testAdd` can not help with this issue, since basic tests (founder and one or no first descendants) and `testAdd` are combined with `&`.

Value

`prune` returns a data.frame with possibly fewer individuals. Read also the details.
### Examples

```r
## Pedigree example
x <- data.frame(oseba=c(1, 9, 11, 2, 3, 10, 12, 13, 4, 5, 6, 7, 14, 15, 16, 17),
                oce=c(2, 10, 12, 5, 0, 7, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0),
                mama=c(3, 8, 13, 0, 4, 0, 0, 0, 0, 14, 6, 0, 0, 15, 16, 17, 0),
                spol=c(2, 2, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 1, 1, 1, 1),
                generacija=c(1, 1, 1, 2, 2, 1, 2, 2, 2, 2, 3, 3, 4, 4, 5, 6, 7, 8),
                last=c(2, NA, 8, 4, 1, 6, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA))

## Default case
prune(x=x, id="oseba", father="oce", mother="mama", unknown=0)

## Use of additional test i.e. do not remove individual if it has
## known value for "last"
prune(x=x, id="oseba", father="oce", mother="mama", unknown=0,
      testAdd=is.na(x$last))

## Use of other data
y <- data.frame(oseba=c(11, 15, 16),
                last2=c(8.5, 7.5, NA))

x <- merge(x=x, y=y, all.x=TRUE)
prune(x=x, id="oseba", father="oce", mother="mama", unknown=0,
      testAdd=is.na(x$last2))
```

---

**relationshipAdditive**  
Additive relationship matrix and its inverse

**Description**

relationshipAdditive creates additive relationship matrix, while inverseAdditive creates its inverse directly from a pedigree. kinship is another definition of relationship and is equal to half of additive relationship.

**Usage**

```r
relationshipAdditive(x, sort=TRUE, names=TRUE, ...)
inverseAdditive(x, sort=TRUE, names=TRUE, ...)
kinship(x, sort=TRUE, names=TRUE, ...)
```
Arguments

- **x**: Pedigree
- **sort**: logical, for the computation the pedigree needs to be sorted, but results are sorted back to original sorting (sort=TRUE) or not (sort=FALSE)
- **names**: logical, should returned matrix have row/colnames; this can be used to get leaner matrix
- ... arguments for other methods

Details

Additive or numerator relationship matrix is symmetric and contains $1 + F_i$ on diagonal, where $F_i$ is an inbreeding coefficients (see `inbreeding`) for subject $i$. Off-diagonal elements represent numerator or relationship coefficient between subjects $i$ and $j$ as defined by Wright (1922). Henderson (1976) showed a way to set up inverse of relationship matrix directly. Mrode (2005) has a very nice introduction to these concepts.

Take care with `sort=FALSE, names=FALSE`. It is your own responsibility to assure proper handling in this case.

Value

A matrix of $n \times n$ dimension, where $n$ is number of subjects in x

Author(s)

Gregor Gorjanc and Dave A. Henderson

References


See Also

`Pedigree`, `inbreeding` and `geneFlowT`

Examples

data(Mrode2.1)
Mrode2.1$dtB <- as.Date(Mrode2.1$dtB)
x2.1 <- Pedigree(x=Mrode2.1, subject="sub", ascendant=c("fat", "mot"),
                   ascendantSex=c("M", "F"), family="fam", sex="sex",
                   generation="gen", dtBirth="dtB")

(A <- relationshipAdditive(x2.1))
fractions(A)
solve(A)
inverseAdditive(x2.1)
relationshipAdditive(x2.1[3:6,])
Remove individual from pedigree

Description

removeIndividual provides utility for removing individuals from a pedigree.

Usage

removeIndividual(x, individual, remove="all")

Arguments

x  pedigree
individual  vector of individuals
remove  character, column names of id columns and/or "all", see details

Details

Individuals passed to argument individual will be removed from the pedigree. If there is a pedigree with individual "id" and two ascendants, say "father" and "mother", then one can pass any combination of these three id columns or "all" for all of them in short to argument remove. In case only "id" is passed to remove, individuals will be removed from the pedigree, but not from ascendant id columns, which might be a matter of interest only if specified individuals show up as ascendants for some other individuals. In case you want to remove an individual completely from the pedigree "all" must be used.

Individuals in id column are removed via removal of the whole record from the pedigree. Individuals in ascendant id columns are only replaced by attr(x, "unknown").

If founder is removed, attribute extended status is changed to FALSE.

Author(s)

Gregor Gorjanc

See Also

Pedigree

Examples

ped <- generatePedigree(3)
summary(ped)
removeIndividual(ped, individual=c(1, 3, 4), remove="father")
removeIndividual(ped, individual=c(1, 3, 4), remove=c("mother", "father"))
(ped <- removeIndividual(ped, individual=c(1, 3, 4), remove=c("all")))
summary(ped)
Description
Pedigree sort

Usage

```r
## S3 method for class 'Pedigree'
sort(x, decreasing=FALSE, na.last=TRUE, ..., by="default")
```

Arguments

- `x`: pedigree, object to be sorted
- `decreasing`: logical, sort order
- `na.last`: logical, control treatment of NAs; check `order`
- `...`: arguments passed to `order`, see details
- `by`: character, sort by "default", "pedigree", "generation", or "dtBirth" information, see details

Details

Sorting of the pedigree can be performed in different ways. Since pedigree can contain date of birth, sorting by this would be the most obvious way and it would be the most detailed sort. However, there might be the case that date of birth is not available for some or all individuals. Therefore, this function by default (when `by="default"`) tries to figure out what would be the best way to perform the sort. If date of birth is available for all individuals then date of birth is used for sorting. If not, generation information is used, but only if it is known for all individuals (it should be more or less easy to figure out the generation for all individuals in the pedigree CHECK). Again if not, sorting is done via information in pedigree i.e. ascendants will precede descendants or vice versa. User can always define it's own preference by argument by. When `by="dtBirth"` or `by="generation"` sorting is performed via `order` and its arguments `na.last` and `decreasing` can be used. With `by="pedigree"` argument `decreasing` has an effect.

Generation values can have different meaning i.e. values might either increase or decrease from ascendants to descendant with the same meaning. This information is stored in attribute `generationOrder` (at the time of creating the pedigree object via `Pedigree`) and used for determining the order of sorting if sorting is by generation. The output of the result might therefore be opposite of what user might expect. If that is the case, use argument `decreasing` as defined in `order`. Look also into examples bellow.

Value

Sorted pedigree

Author(s)

Gregor Gorjanc
See Also

Pedigree and order

Examples

```r
ped <- generatePedigree(nId=5)
ped <- ped[sample(1:nrow(ped)), ]
sort(ped)
## sort(ped, by="dtBirth")  ## TODO
## try(sort(ped, by="pedigree"))  ## TODO

## Sorting with decreasing generation values from ascendants to descendants
ped1 <- generatePedigree(nId=5, generationOrder="decreasing")
sort(ped1, by="generation")
sort(ped1, decreasing=TRUE, by="generation")
sort(ped1, decreasing=FALSE, by="generation")

## Sorting with unknown values
ped[1, "generation"] <- NA
sort(ped, na.last=TRUE, by="generation")
sort(ped, na.last=FALSE, by="generation")
sort(ped, na.last=NA, by="generation")
```

summary.Pedigree

**Pedigree summary**

Description

summary.Pedigree reports TODO.

Usage

```r
## S3 method for class 'Pedigree'
summary(object, ...)
```

Arguments

- **object**
  - pedigree object

- ... additional arguments for other methods (not used)

Details

TODO.

Value

TODO.

Author(s)

Gregor Gorjanc
See Also

Pedigree

Examples

```r
ped <- generatePedigree(nId=5)
summary(ped)
```

<table>
<thead>
<tr>
<th>undocumented</th>
<th>Undocumented functions</th>
</tr>
</thead>
</table>

Description

These functions are undocumented. Some are internal and not intended for direct use. Some are not yet ready for end users. Others simply haven’t been documented yet.
Index

*Topic array
  geneContribution, 6
  geneFlowT, 7
  inbreeding, 15
  relationshipAdditive, 26
*Topic attribute
  check, 2
*Topic datasets
  Mrode, 19
*Topic manip
  extend, 4
  generatePedigree, 9
  Pedigree, 21
  prune, 24
  summary.Pedigree, 30
*Topic misc
  check, 2, 22, 23
  checkId(check), 2
  codes, 22
  expectedGenotypes, 15
  extend, 4, 22, 23
  Falconer (Mrode), 19
  Falconer5.1 (Mrode), 19
  family, 4, 5, 22
  family<- (family), 5
  gameteFlowM(geneFlowT), 7
  geneContribution, 6
  geneFlowT, 7, 17, 27
  geneFlowTinv (geneFlowT), 7
  generatePedigree, 9
  generation, 10, 22
  generation<- (generation), 10
  genotype, 15
  gpi, 11, 14, 15
  gpLong2Wide, 12, 13
  gpLong2Wide(hwp), 14
  hwp, 12, 13, 14
  inbreeding, 8, 15, 27
  inverseAdditive, 18
  inverseAdditive (relationshipAdditive), 26
  is.Pedigree (Pedigree), 21
  isFounder, 17
  kinship, 8, 17
  kinship (relationshipAdditive), 26
  mendelianSamplingD (geneFlowT), 7
  merge, 25
  model.matrix, 18
  model.matrix.Pedigree, 18
  Mrode, 19
  Mrode2.1 (Mrode), 19
  Mrode3.1 (Mrode), 19
  nIndividual, 20

as.character.Pedigree (Pedigree), 21
as.factor.Pedigree (Pedigree), 21
as.integer.Pedigree (Pedigree), 21
as.Pedigree (Pedigree), 21

check, 2, 22, 23
checkId(check), 2
codes, 22
equivalentGenotypes, 15
extend, 4, 22, 23
Falconer (Mrode), 19
Falconer5.1 (Mrode), 19
family, 4, 5, 22
family<- (family), 5
gameteFlowM(geneFlowT), 7
geneContribution, 6
geneFlowT, 7, 17, 27
geneFlowTinv (geneFlowT), 7
generatePedigree, 9
generation, 10, 22
generation<- (generation), 10
genotype, 15
gpi, 11, 14, 15
gpLong2Wide, 12, 13
gpLong2Wide (hwp), 14
hwp, 12, 13, 14
inbreeding, 8, 15, 27
inverseAdditive, 18
inverseAdditive (relationshipAdditive), 26
is.Pedigree (Pedigree), 21
isFounder, 17
kinship, 8, 17
kinship (relationshipAdditive), 26
mendelianSamplingD (geneFlowT), 7
merge, 25
model.matrix, 18
model.matrix.Pedigree, 18
Mrode, 19
Mrode2.1 (Mrode), 19
Mrode3.1 (Mrode), 19
nIndividual, 20
INDEX

order, 29, 30

Pedigree, 3, 4, 6–10, 17, 18, 20, 21, 26–31
prune, 24

relationshipAdditive, 8, 17, 18, 26
removeIndividual, 28

sort, 22, 23
sort.Pedigree, 29
summary.Pedigree, 30

undocumented, 31