Package ‘HyRiM’

April 23, 2020

Type Package

Title Multicriteria Risk Management using Zero-Sum Games with Vector-Valued Payoffs that are Probability Distributions

Version 2.0.0

Date 2020-04-10

Imports compare, orthopolynom, grImport2, Rglpk, purrr

Author Stefan Rass, Sandra Koenig, Ali Alshawish

Maintainer Austrian Institute of Technology <hyrim@ait.ac.at>

Description

Construction and analysis of multivalued zero-sum matrix games over the abstract space of probability distributions, which describe the losses in each scenario of defense vs. attack action. The distributions can be compiled directly from expert opinions or other empirical data (insofar available). The package implements the methods put forth in the EU project HyRiM (Hybrid Risk Management for Utility Networks), FP7 EU Project Number 608090. The method has been published in Rass, S., König, S., Schauer, S., 2016. Decisions with Uncertain Consequences-A Total Ordering on Loss-Distributions. PLoS ONE 11, e0168583. <doi:10.1371/journal.pone.0168583>, and applied for advanced persistent thread modeling in Rass, S., König, S., Schauer, S., 2017. Defending Against Advanced Persistent Threats Using Game-Theory. PLoS ONE 12, e0168675. <doi:10.1371/journal.pone.0168675>. A volume covering the wider range of aspects of risk management, partially based on the theory implemented in the package is the book edited by S. Rass and S. Schauer, 2018. Game Theory for Security and Risk Management: From Theory to Practice. Springer, <doi:10.1007/978-3-319-75268-6>, ISBN 978-3-319-75267-9.

NeedsCompilation no

Encoding UTF-8

Suggests knitr,rmarkdown,qpdf,testthat

Repository CRAN

Date/Publication 2020-04-23 08:50:02 UTC

R topics documented:

`HyRiM-package` *Multicriteria Risk Management using Zero-Sum Games with Vector-Valued Payoffs that are Probability Distributions*

Description

Construction and analysis of multivalued zero-sum matrix games over the abstract space of probability distributions, which describe the losses in each scenario of defense vs. attack action. The distributions can be compiled directly from expert opinions or other empirical data (insofar available). The package implements the methods put forth in the EU project HyRiM (Hybrid Risk Management for Utility Networks), FP7 EU Project Number 608090. The method has been published in Rass, S., KÃ¶nig, S., Schauer, S., 2016. Decisions with Uncertain Consequences-A Total Ordering on Loss-Distributions. PLoS ONE 11, e0168583. <doi:10.1371/journal.pone.0168583>, and applied for advanced persistent thread modeling in Rass, S., KÃ¶nig, S., Schauer, S., 2017. Defending Against Advanced Persistent Threats Using Game-Theory. PLoS ONE 12, e0168675. <doi:10.1371/journal.pone.0168675>. A volume covering the wider range of aspects of risk management, partially based on the theory implemented in the package is the book edited by S. Rass and S. Schauer, 2018. Game Theory for Security and Risk Management: From Theory to Practice. Springer, <doi:10.1007/978-3-319-75268-6>, ISBN 978-3-319-75267-9.

Author(s)

Stefan Rass, Sandra Koenig, Ali Alshawish

Maintainer: Austrian Institute of Technology <hyrim@ait.ac.at>

References

S. Rass, S. Koenig, S. Schauer: Uncertainty in Games: Using Probability-Distributions as Payoffs. in MHR Khouzani et al. (Eds.) GameSec 2015, Springer LNCS 9406, pp. 346-357, DOI: 10.1007/978-3-319-25594-1_20.

S. Rass. On Game-Theoretic Risk Management (Part One). Towards a Theory of Games with Payoffs that are Probability-Distributions. ArXiv e-prints, June 2015. http://arxiv.org/abs/1506.07368.

S. Rass. On Game-Theoretic Risk Management (Part Two). Algorithms Algorithms to Compute Nash-Equilibria in Games with Distributions as Payoffs, ArXiv e-prints, arXiv:1511.08591, 2015.

`cdf` *(cumulative) loss distribution function*

Description

returns the numeric values of the cumulative loss distribution `ld` evaluated at `x`, i.e., $\Pr(X \leq x)$, where $X \sim \text{ld}$.

Usage

```
cdf(ld, x)
```

Arguments

`ld`	the loss distribution as obtained from `lossDistribution` or `mgss`.
`x`	the point at which the distribution function shall be evaluated (must be a numeric; vectors are not supported yet)

Details

the function internally distinguishes discrete and continous disributions only in terms of rounding its argument to the largest integer less than x. Its value is obtained by numeric integration of the internal representation of the loss distribution (in the continuous case).

For discrete distributions, the function works on the internal probability mass function (which may be different from the empirical distribution in case that the loss distribution has been smoothed during its construction; see `lossDistribution`).

Value

an approximation for the probability Pr(ld<=x).

Note

in its current version, `cdf` does not vectorize, i.e., cannot be applied to vector arguments `x`.

Author(s)

Stefan Rass

See Also

suitable inputs for this function are provided by `lossDistribution` and `mgss`.

Examples

```
cvss1base <- c(10,6.4,9,7.9,7.1,9)
ld <- lossDistribution(cvss1base)
cdf(ld, 4)
```

`disappointmentRate` *computation of the disappointment rate*

Description

For a minimizing player, the *disappointment rate* is the likelihood for the loss to exceed its expectation (thus disappoint the defender). For any random loss X, it is given by $Pr(X > E(X))$.

Usage

```
disappointmentRate(d, x, y, verbose = TRUE, ...)
```

Arguments

`d`	a lossDistribution object or a matrix; typically the assurance from a previously computed equilibrium (see `mgss`). In that case, all other parameters are ignored. Alternatively, one can provide a matrix of real values instead, to compute the disappointment rate in the so-specified zero-sum matrix game. In that case, the other parameters are also taken into considertion.
`x,y`	the mixed strategies under which the disappointment rate shall be computed. Usually, this would be an equilibrium of the (real-valued) matrix game. If only `x` or only `y` is supplied, the function computes a best response to the given (mixed) strategy. If both are omitted, the function internally computes an equilibrium by a call to `mgss`.
`verbose`	if set to `FALSE`, suppresses all messaging.
`...`	further parameters internally passed onwards to `mgss` to compute an equilibrium.

Details

The disappointment rate can be taken as an auxiliary goal to optimize, though it is not supported for optimization in the current version of the package. Note that it does not make sense to consider this rate as an isolated (single) goal, since the optimal strategy would then be playing towards maximal losses (with explicit aid of the opponent) in order to minimize the mass to the left of the expected loss. However, it is a quantity of interest when the equilibrium has been computed, as it indicates how "satisfying" the equilibrium will be upon playing.

Value

the likelihood to overshoot the expectation of the random loss X with distribution `d`, i.e., $Pr(X > E(X))$.

Author(s)

Stefan Rass

References

see for example, F. Gul: "A Theory of Disappointment Aversion", Econometrica, vol. 59, no. 3, p. 667, 1991.

See Also

mgss

Examples

```
library(compare)
library(orthopolynom)
## raw data (PURELY ARTIFICIAL, for demo purposes only)
# N=100 observations in each category
obs111<-c(rep(1,40),rep(3,20),rep(5,10),rep(7,20),rep(9,10));
obs112<-c(rep(1,50),rep(2,10),rep(4,10),rep(6,20),rep(8,10));
obs121<-c(rep(1,20),rep(4,30),rep(6,20),rep(8,10),rep(10,20));
obs122<-c(rep(1,40),rep(2.5,20),rep(5,20),rep(7.5,10),rep(9,10));
obs211<-c(rep(1,30),rep(2,30),rep(5,10),rep(8,10),rep(10,20));
obs212<-c(rep(1,10),rep(2,10),rep(4,20),rep(7,20),rep(10,40));
obs221<-c(rep(1,30),rep(3,30),rep(4,10),rep(7,20),rep(9,10));
obs222<-c(rep(1,10),rep(3,10),rep(5,50),rep(8,20),rep(10,10));
obs311<-c(rep(1,40),rep(2,30),rep(4,10),rep(7,10),rep(9,10));
obs312<-c(rep(1,20),rep(3,20),rep(4,20),rep(7,20),rep(10,20));
obs321<-c(rep(1,10),rep(3,40),rep(4,30),rep(7,10),rep(9,10));
obs322<-c(rep(1,10),rep(4,30),rep(5,30),rep(7,10),rep(10,20));

## compute payoff densities
f111<-lossDistribution(obs111)
f112<-lossDistribution(obs112)
f121<-lossDistribution(obs121)
f122<-lossDistribution(obs122)
f211<-lossDistribution(obs211)
f212<-lossDistribution(obs212)
f221<-lossDistribution(obs221)
f222<-lossDistribution(obs222)
f311<-lossDistribution(obs311)
f312<-lossDistribution(obs312)
f321<-lossDistribution(obs321)
f322<-lossDistribution(obs322)

payoffs<-list(f111,f112,f121, f122,f211,f212,f221,f222, f311,f312,f321,f322)
G <- mosg( n=2,
            m=2,
            payoffs,
            goals=3,
            goalDescriptions=c("g1", "g2", "g3"),
            defensesDescr = c("d1", "d2"),
```

```
            attacksDescr = c("a1", "a2"))
eq <- mgss(G,weights=c(0.25,0.5,0.25))

# get the disappointment rate for the first security goal g1
disappointmentRate(eq$assurances$g1)

############
# construct a game with one goal and related disappointment
payoffs <- list(f111,f112,f121,f122)
# note that from here onwards, the code is "generic", meaning that
# exactly the same procedure would apply to *any* kind of game that
# we want to play with disappointments, as long as the input data comes
# in the variable "payoffs" (as used in the code below)
expectations <- unlist(lapply(payoffs, mean))
disappointmentRates <- unlist(lapply(payoffs, disappointmentRate))
# put the two goals together in a game
gameWithDisappointment <- c(expectations, disappointmentRates)
G <- mosg( n=2,
          m=2,
          losses=gameWithDisappointment,
          goals=2,
          goalDescriptions=c("revenue", "disappointment"),
          defensesDescr = c("d1", "d2"),
          attacksDescr = c("a1", "a2"))
eq <- mgss(G,weights=c(0.1,0.9))
```

`lossDistribution` *construction and handling of loss distributions*

Description

Loss distributions can be constructed from both, continuous and categorical data. In any case, the input data must be a list (vector) of at least two numeric values all being ≥ 1. For discrete data, the function additionally takes the full range of categories, all being represented as integers (with the lowest category having the number 1).

Usage

```
# construct a loss distribution from data
lossDistribution(
  dat,
  discrete = FALSE,
  dataType = c("raw", "pdf", "cdf"),
  supp = NULL,
  smoothing = c("none", "ongaps", "always"),
  bw = NULL)
# get information about the loss distribution
## S3 method for class 'mosg.lossdistribution'
```

```
print(x, ...)
## S3 method for class 'mosg.lossdistribution'
summary(object, ...)
## S3 method for class 'summary.mosg.lossdistribution'
print(x, ...)
## S3 method for class 'mosg.lossdistribution'
plot(x, points = 100, xlab = "", ylab = "",
           main = "", p = 0.999, newPlot = TRUE, cutoff = NULL, ...)
# get quantiative information about the distribution
## S3 method for class 'mosg.lossdistribution'
quantile(x, p, eps = 0.001, ...)
## S3 method for class 'mosg.lossdistribution'
mean(x, ...)
# evaluate the loss density function
## S3 method for class 'mosg.lossdistribution'
density(x, t, ...)
# for the cumulative distribution function, see the function 'cdf'
```

Arguments

`dat`	a vector of at least two input observations (all ≥ 1 required)
`discrete`	defaults to `FALSE`. If set to `TRUE`, the loss distribution is constructed as discrete. In that case, a value for supp is required.
`dataType`	applies only if `discrete=TRUE`, and specifies how the values in `dat` are to be interpreted. Defaults to `raw`, by which the data is taken as observations. Given as `pdf`, the values in `dat` are directly interpreted as a probability density (checked for nonnegativity and re-normalized if necessary). If the data type is specified as `cdf`, then the values in `dat` are taken as cumulative distribution function, i.e., checked to be non-decreasing, non-negative and re-normalized to 1 if necessary.
`supp`	if the parameter `discrete` is set to `TRUE`, then this parameter must be set as a vector of two elements, specifying the minimal and maximal category, e.g. `supp=c(1,5)`.
`bw`	the bandwidth parameter (numeric value) for kernel smoothing. Defaults internally to the result of bw.nrd0 if omitted.
`x`	a loss distribution object returned by `lossDistribution` or `mgss`, or a value within the support of a loss distribution.
`t`	a value within the support of `ld` or a summary object for a loss distribution.
`object`	a loss distribution object
`eps`	the accuracy at which the quantile is approximated (see the details below).
`smoothing`	string; partially matched with "none" (default), "ongaps", and "always". If set to "always", then the function computes a discrete kernel density estimate (using a discretized version of a Gaussian density with a bandwidth as computed by `bw.nrd0` (Silverman's rule)), to assign categories with zero probability a positive likelihood. If set to "ongaps", then the smoothing is applied only if necessary (i.e., if the probability mass is zero on at least one category). the function `plot.mosg.lossdistribution` takes the parameters:

`points`	the number of points at which loss densities are is evaluated (numerically) for plotting.
`xlab`	a label for the x-axis in the plot.
`ylab`	a label for the y-axis in the plot.
`main`	a title for the plot
`p`	a quantile that determines the plot range for the loss distribution
`newPlot`	if set to `TRUE`, then a new plot is opened. Otherwise, the plot is added to the current plot window (typcially used by `plot.mosg` to visualize game matrices).
`cutoff`	the cutoff point at which all densities shall be truncated before plotting (note that the mass functions are rescaled towards unit mass).
`...`	further arguments passed to or from other methods

Details

The function internally computes a Gaussian kernel density estimator (KDE; using Silverman's rule of thumb for the bandwidth selection) on the continuous data. The distribution is truncated at the maximal observation supplied + 5*the bandwidth of the Gaussian KDE, or equivalently, at the right end of the support in case of discrete distributions.

For discrete distributions, missing observations are handled by smoothing the density (by convolution with a discretized Gaussian kernel). As an alternative, a re-definition of categories may be considered.

Degenerate distributions are not supported! The construction of classical games with real-valued payoffs works directly through mosg by supplying a list of values rather than loss distributions. See the example given with mosg.

The generic functions `quantile`, `mean` and `density` both distinguish discrete from continuous distributions in the way of how values are being computed.

Quantiles are computed using the direct definition as an approximation `y` so that x = Pr(ld <= y). For continuous distributions, a bisective search is performed to approximate the inverse cumulative distribution function. For discret distributions, `quantile` works with cumulative sums. The accuracy parameter `eps` passed to `quantile` causes the bisective search to stop if the search interval has a length less than `eps`. In that case, the middle of the interval is returned. For discrete distributions, the computation is done by cumulative sums on the discrete probability mass function.

`mean` either invokes `moment(ld,1)` to compute the first moment.

`density` is either a wrapper for the internal representation by the `function` object `lossdistr`, or directly accesses the probability mass function as internally stored in the field `dpdf` (see the 'values' section below).

For visualization, `plot` produces a bar plot for categorical distributions (over categories as specified by the `supp` field; see the 'values' section below), and for continous distributions, a continuous line plot is returned on the range 1...max(`range` + 5*`bw`), where the values are described below. To ease comparison and a visual inspection of the game matrix, the default plot ranges can be overridden by supplying `xlim` and `ylim` for the plot function.

Value

The return values of `lossDistribution` is an object of class `mosg.lossdistribution`. The same goes for `lossDistribution.mosg`.

- `observations` carries over the data vector supplied to construct the distribution.
- `range` the minimal and maximal loss observed, as a 2-element vector. For loss distributions induced by games, the range is the smallest interval covering the ranges of all distributions in the game.
- `bw` the bandwidth used for the kernel density approximate.
- `lossdistr` a `function` embodying the kernel density (probability mass function) as a spline function (for continuous densities only)
- `normalizationFactor` the factor by which `lossdistr` must be multiplied (to normalize under the truncation at max(observations) + 5*bw.
- `is.mixedDistribution` a flag indicating whether or not the distribution was constructed by a call to `lossDistribution` or the generic function `lossDistribution.mosg`.
- `is.discrete` a flag set to `TRUE` if the distribution is over categories
- `dpdf` if `is.discrete` is `TRUE`, then this is a vector of probability masses over the support (field `supp`).
- `supp` if `is.discrete` is `TRUE`, then this is a 2-element vector specifying the minimal and maximal loss category (represented by integers).

A `summary` returns an object of class `mosg.equilibrium.summary`, for which the generic `print` function can be applied, and which carries the following fields:

- `range` the minimal and maximal observation of the underlying data (if available), or the minimal and maximal losses anticipated for this distribution (e.g., in case of discrete distributions the common support).
- `mean` the first moment as computed by `mean`.
- `variance` the variance as computed by `variance`.
- `quantiles` a 2x5-matrix of quantiles at levels 10%, 25%, 50%, 75% and 90%.

Note

If the plotting throws an error concerning too large figure margins, then adjusting the plot parameters using par may help, since the plot function does not override any of the current plot settings (e.g., issue `par(c(0,0,1,1) + 0.1))` before plotting to reduce the spacing close towards zero))

In some cases, plots may require careful customization to look well, so playing aroud with the other settings as offered by par can be useful.

If the distribution has been smoothed, then `mean`, `variance`, `quantile`, `density` and `cdf` will refer to the smoothed version of the distribution. In that case, the returned quantities are mere approximations of the analogous values obtained directly from the underlying data.

Author(s)

Stefan Rass

See Also

mosg, mgss, cdf, variance

Examples

```
# construct a loss distribution from observations (raw data)
cvss1base <- c(10,6.4,9,7.9,7.1,9)
ld <- lossDistribution(cvss1base)
summary(ld)
plot(ld)

# construct a loss distribution of given shape
# for example, a Poisson density with lambda = 4
x <- 1:10
f <- dpois(x, lambda = 4)
# construct the loss distribution by declaring the data
# to be a probability density function (pdf)
ld <- lossDistribution(f, dataType = "pdf", discrete = TRUE, supp = range(x))
# note that this call throws a warning since it internally
# truncates the loss distribution to the support 1:10, and
# renormalizes the supplied density for that matter.

# for further examples, see the documentation to 'mosg' and 'mosg.equilibrium'
```

---

mgss — *compute a multi-goal security strategy*

Description

Finds security strategy that assures a maximal loss w.r.t. all goals of the given game, delivering a Pareto-efficient loss bound. Internally, it constructs an auxiliary one-against-all game and uses a sequence of linear programs to compute a Pareto-Nash equilibrium therein (Rass, 2013), using the methods described by (Lozovanu et al 2005; Rass 2015).

Usage

```
mgss(G, weights, cutOff, ord = 5, fbr = FALSE, points = 512)
```

Arguments

`G`	a multi-objective game constructed using `mosg`
`weights`	each goal in `G` can be assigned a weight to reflect its priority. If missing, the weights default to be all equal. The weights do not need to sum up to 1 (and are normalized towards a unit sum otherweise), but need to be all non-negative.
`cutOff`	(only used for continuous loss distributions) the maximal loss for which no events are expected or otherwise the risk of exceeding `cutOff` are accepted. If missing, this value defaults to the maximal observation on which the loss distributions were constructed (equivalently, the right end of their common support).

`ord` the order up to which a continuous loss distribution shall be approximated. This value may be set to high orders when it is necessary to distinguish distributions that are similar at the tails.

`fbr` if set to `TRUE`, instruct the function to additionally compute the best replies regarding each goal individually, assuming that defender plays `optimalDefense` as a leader, and the attacker per goal follows (follower's best reply). These replies are always pure strategies.

`points` the number of points at which the resulting equilibrium loss distributions are evaluated numerically.

Details

For continuous loss distributions, the function uses a Gaussian kernel density approximation (constructed using the function `lossDistribution`), and computes a Taylor-polynomial approximation at the x equal to `cutOff` for each distribution up to order `ord`. Preferences are decided using the method put forth in (Rass, König, Schauer, 2016), using sign-alternating derivatives, representing a distribution by a vector with `ord` elements. Categorical distributions are represented likewise directly by the vector of their probability masses. In both cases, Theorem 3 and Lemma 4 in (Rass, König, Schauer 2016) allow a decision about the stochastic order between two distributions by a lexicographic comparison of the order between the vector-representation. Thus, the computed optima are lex-order optimal. Constructing a game using `mosg` with vectors in the payoff description can, consequently, allows to use `mgss` to compute lex-order optimal equilibria for multi-criteria games.

Value

An object of class `mosg.equilibrium`, containing the following fields:

`optimalDefense` a discrete probability distribution over the action space of player 1 (defender)

`optimalAttacks` a discrete probability distribution over the action space of player 2 (attacker). Note that this is *not* a best-response to the player 1's `optimalDefense`, but rather the best that the attacker could do if the game were *just about the particular goal* that the attacker refers to. This worst-case scenario assumes that the defender would focus all its efforts to that single goal.

`assurances` a list of loss distributions valid under the assumption that player 1 adheres to the `optimalDefense` distribution in its randomized action choices, while the opponent plays its own zero-sum equilibrium strategy in the game that is only (and exclusively) about this particular goal. This value has to be interpreted with care, as it assumes that player 1 would put all efforts into a defense for the particular goal, but in reality, will have multiple criteria to simultaneously optimize. This means that the attacker, in turn, could adapt to the `optimalDefense` of player 1, to cause more damage. The given assurance is thus only an upper bound of the worst-possible damage, under the assumption that player 1 would focus only on this particular goal.

The list can be accessed by the names for each goal as specified through the input `mosg` object `G`. Each distribution within `assurances` is a mixed loss distribution constructed using `lossDistribution`

br_to_optimalDefense
: This is a vector of best replies per goal for a leading defender playing the fixed strategy optimalDefense, and letting the adversary (player 2) follow. It is the (stochastically largest) damage among $optimalDefense^T \cdot A_p$, when A_p is the game structure for the p-th goal; the vector br_to_optimalDefense contains the indices of the individually best replies, pointing into the list of attack strategies.

Note

The output loss distributions (accessible by the list assurances) cannot be used to construct a subsequent game (see mosg), since continuous distributions are represented as a sequence of points, rather than raw data or probability masses.

As of version 2.0.0 of the package, this function is no longer downwards compatible to earlier versions of itself, since the method of computation (formerly fictitious play) was replaced by linear programming to give exact solutions rather than approximations. Consequently, the parameters T (iteration count) and eps (accuracy) have become useless and have been removed after version 1.0.4.

Author(s)

Sandra Koenig, Stefan Rass

References

S. Rass, S. König, S. Schauer. Decisions with Uncertain Consequences-A Total Ordering on Loss-Distributions. PLoS ONE 11, e0168583. 2016, https://doi.org/10.1371/journal.pone.0168583

S. Rass. On Game-Theoretic Risk Management (Part One). Towards a Theory of Games with Payoffs that are Probability-Distributions. June 2015. http://arxiv.org/abs/1506.07368.

S. Rass. On Game-Theoretic Risk Management (Part Two). Algorithms to Compute Nash-Equilibria in Games with Distributions as Payoffs, 2015, arXiv:1511.08591v1 [q-fin.EC].

D. Lozovanu, D. Solomon, and A. Zelikovsky. Multiobjective games and determining pareto-nash equilibria. Buletinul Academiei de Stiinte a Republicii Moldova Matematica, 3(49):115-122, 2005. ISSN 1024-7696.

See Also

A brief info on the results can be obtained by print.mosg.equilibrium, and a more detailed summary (showing all loss distributions in detail) is obtained by summary.mosg.equilibrium.

Examples

```
library(compare)
library(orthopolynom)
## raw data (PURELY ARTIFICIAL, for demo purposes only)
# N=100 observations in each category
obs111<-c(rep(1,40),rep(3,20),rep(5,10),rep(7,20),rep(9,10));
obs112<-c(rep(1,50),rep(2,10),rep(4,10),rep(6,20),rep(8,10));
obs121<-c(rep(1,20),rep(4,30),rep(6,20),rep(8,10),rep(10,20));
```

```
obs122<-c(rep(1,40),rep(2.5,20),rep(5,20),rep(7.5,10),rep(9,10));
obs211<-c(rep(1,30),rep(2,30),rep(5,10),rep(8,10),rep(10,20));
obs212<-c(rep(1,10),rep(2,10),rep(4,20),rep(7,20),rep(10,40));
obs221<-c(rep(1,30),rep(3,30),rep(4,10),rep(7,20),rep(9,10));
obs222<-c(rep(1,10),rep(3,10),rep(5,50),rep(8,20),rep(10,10));
obs311<-c(rep(1,40),rep(2,30),rep(4,10),rep(7,10),rep(9,10));
obs312<-c(rep(1,20),rep(3,20),rep(4,20),rep(7,20),rep(10,20));
obs321<-c(rep(1,10),rep(3,40),rep(4,30),rep(7,10),rep(9,10));
obs322<-c(rep(1,10),rep(4,30),rep(5,30),rep(7,10),rep(10,20));

## compute payoff densities
f111<-lossDistribution(obs111)
f112<-lossDistribution(obs112)
f121<-lossDistribution(obs121)
f122<-lossDistribution(obs122)
f211<-lossDistribution(obs211)
f212<-lossDistribution(obs212)
f221<-lossDistribution(obs221)
f222<-lossDistribution(obs222)
f311<-lossDistribution(obs311)
f312<-lossDistribution(obs312)
f321<-lossDistribution(obs321)
f322<-lossDistribution(obs322)

payoffs<-list(f111,f112,f121, f122,f211,f212,f221,f222, f311,f312,f321,f322)
G <- mosg( n=2,
            m=2,
            payoffs,
            goals=3,
            goalDescriptions=c("g1", "g2", "g3"),
            defensesDescr = c("d1", "d2"),
            attacksDescr = c("a1", "a2"))
eq <- mgss(G,weights=c(0.25,0.5,0.25))
print(eq)
summary(eq)

# construct another loss distribution from a given behavior in the game G
suboptimal <- lossDistribution.mosg(G, c(0.1,0.1,0.8), c(0.2,0.3,0.5))
plot(suboptimal)

# compute an equilibrium in a standard matrix game
#     [,1] [,2]
#[1,]    3    4
#[2,]    6    1
G <- mosg(n = 2, m = 2, goals = 1,
          losses = list(3,6,4,1), byrow=FALSE,
          attacksDescr = c("a1", "a2"))
mgss(G, fbr=TRUE)  # compute an equilibrium, including best replies if the adversary is a follower

# get best replies if there would be a following
# adversary per goal (taking the defender as a leader)
G$attacksDescriptions[eq$br_to_optimalDefense]
```

`moment` *compute moments of loss distributions*

Description

the moment of given order k is computed by numeric integration or summation (in case of discrete distributions)

Usage

```
moment(ld, k)
```

Arguments

`ld`	the loss distribution as obtained from `lossDistribution` or `mgss`.
`k`	the order of the moment (must be an integer ≥ 1)

Value

the k-th order moment of the given loss distribution

Note

In case of continuous distributions, the value returned is an approximation and based on the internal kernel density approximation.

For categorical distributions, the function works on the internal probability mass function (which may be different from the empirical distribution in case that the loss distribution has been smoothed during its construction; see `lossDistribution`).

In its current version, `cdf` does not vectorize, i.e., cannot be applied to vector arguments `x`.

Author(s)

Stefan Rass

See Also

the methods `mean` and `variance` are based on this function.

Examples

```
cvss1base <- c(10,6.4,9,7.9,7.1,9)
ld <- lossDistribution(cvss1base)
cdf(ld, 4)
```

---

mosg *Construction and handling of multi-objective security games*

Description

this function takes a list of loss distributions construced using `lossDistribution`, along with a specification of the game's shape (number of strategies for both players and number of goals for the first player), and returns an object suitable for analysis by `mgss` to compute a multi-goal security strategy.

Usage

```
mosg( n,
      m,
      goals,
      losses,
      byrow = TRUE,
      goalDescriptions = NULL,
      defensesDescr = NULL,
      attacksDescr = NULL)

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

## S3 method for class 'mosg'
plot(x,
     goal = 1,
     points = 100,
     cutoff = NULL,
     largeGame = FALSE,
     subPlotWidth = 2,
     subPlotHeight = 2,
     cleanUp = TRUE, ...)

# construct a loss distribution by playing a given strategy in the game G
## S3 method for class 'mosg'
lossDistribution(G, player1Strat, player2Strat, points = 512, goal = 1)
```

Arguments

`n`	number of defense strategies (cardinality of the action space for player 1)
`m`	number of attack stratgies (cardinality of the action space for player 2)
`goals`	number of goals for player 1 (must be ≥ 1)
`losses`	a `list` with `n*m*goals` entries, which specifies a total of `goals` game matrices, each with shape `n`-by-`m`. The way in which the game matrices are filled from this

list is controlled by the parameter byrow. Note that in every case, it is assumed that one matrix is specified after the other in the list.

Furthermore, the function assumes all loss distributions having a common support. This is only explicitly verified for discrete distributions (with errors reported), but implicitly assumed to hold for continuous distributions without further checks.

Typically, a game will be constructed from a list of loss distributions obtained by invocations of lossDistribution.

Games can be defined with real-valued (scalar) payoffs if a list of numbers is provided instead. Internally, the function converts these numbers into Bernoulli distributions; a scalar payoff a is converted into a Bernoulli random variable X having $\Pr(X = a) = p \propto a$. This conversion is equivalent to an invocation of lossDistribution with the parameters dat=c(1-p,p), discrete=TRUE, dataType="pdf", smoothing="none", bw = 1 and supp=c(1,2).

If the list of losses comes as a list of vectors, mosg will construct a game assuming a lexicographic order on the loss vectors (with the order being determined from left to right along the coordinates). To this end, mosg checks for all loss vectors to have the same length (otherwise, an error is reported). Negative and zero values in the loss vector *are allowed*.

byrow
: by default (TRUE), the game matrices are filled row-by-row from list losses. If set to FALSE, then the game matrices are filled column-by-column.

goalDescriptions
: if specified, this can be any vector (e.g., textual descriptions) for the goals. Defaults to 1, 2, 3, ... if missing. The length must be equal to goals.

defensesDescr
: if specified, this can be any vector (e.g., textual descriptions) for the defense strategies. Defaults to 1, 2, 3, ... if missing. The length must be equal to n.

attacksDescr
: if specified, this can be any vector (e.g., textual descriptions) for the attack strategies. Defaults to 1, 2, 3, ... if missing. The length must be equal to m.

for the functions print, summary and plot

x
: a game, object of class "mosg", as constructed by the function mosg

The function plot additionally takes the following parameters:

goal
: an integer referring to the goal of interest (for plotting or to construct a loss distribution for). Defaults to the first goal if omitted.

points
: The number of points at which the density is evaluated (for continuous losses); this parameter is ignored for categorical losses.

cutoff
: the cutoff point at which all densities shall be truncated before plotting (note that the mass functions are rescaled towards unit mass). The plot function overrides the following settings internally (so supplying these as parameters will raise an error): xlab,ylab,main,type,names.arg and font.main (applying differently for bar and line plots)

largeGame
: if the plot exits with the error "figure margins too large", one can set this parameter to TRUE, causing plot to write to a temporary SVG file (scalable vector graphics), to avoid the figure space issue and hence the error. The price is a (potentially much) slower plotting, since the system creates the file, and loads it afterwards from the harddisk (cleaning up the file after displaying it).

The size of the plot is controllable by setting the parameters `subPlotWidth` and `subPlotHeight`, see below.

`subPlotWidth` the width in inches for each payoff distribution in the game matrix. This parameter is ignored when `largeGame` is set to FALSE (the default).

`subPlotHeight` the height in inches for each payoff distribution in the game matrix. This parameter is ignored when `largeGame` is set to FALSE (the default).

`cleanUp` If the graph is to be used in other programs, one can supply `cleanUp = FALSE` to retain the temporary SVG file for subsequent use and prints a message where to find the file. By default, the temporary file gets deleted.

The function `lossDistribution.mosg` can be used to play any (given) strategies for player 1 and player 2, and compute the resulting loss from the game.

`G` a game constructed by `mosg` to deliver the loss distribution through its game matrices.

`player1Strat` a discrete distribution over the action space for the defending player 1 in the game `G`

`player2Strat` a discrete distribution over the action space for the attacking player 2 in the game `G`

`...` further arguments passed to or from other methods

Details

Upon input, the function does some consistency checks, such as testing the length of the parameter `losses` to be equal to n*m*goals. The loss distributions are checked for mutual consistency in terms of all being continuous or all being discrete (a mix is not allowed), and all being not mixed distributions (that is, the output distribution of a previous call to `mgss` cannot be used as input to this function).

The functions `print.mosg` gives a brief overview of the game, listing only the shape and strategies for both players. For detailed information, use `summary` on a specific loss distribution in the list for the game (field `losses`).

For plotting games, `plot.mosg` constructs an (n x m)-matrix of loss distributions with rows and columns in the grid being labeled by the values in `defensesDescr` and `attacksDescr`. The plot heading is the name for the specified goal. The function makes no changes to the plot parameters, so fine tuning can be done by changing the settings using the par function.

The function `lossDistribution.mosg` can be used to compute the distribution $x^T * A * y$, for the payoff distribution matrix A, and mixed strategies x (`player1strat`) and y (`player2strat`) in the game. The computation is by a pointwise addition of loss distributions, with the number of points being specifiable by the parameter `points`, which defaults to 512.

Value

The function returns an object of class `mosg`, usable with the function `mgss` to determine a security strategy (i.e., an equilibrium assuming a zero-sum one-against-all competition). The fields returned in the `mosg` object are filled with the input values supplied. In detail, the fields are:

`nDefenses` the value of the parameter `n`

`nAttacks` the value of the parameter `m` `dim` the value of the parameter `goals`

`attacksDescriptions`, `defensesDescriptions`, `goalDescriptions`
: if supplied, then these are filled with the values of `goalDescr`, `defensesDescr` and `attacksDescr`; otherwise, they contain the default values described above.

`maximumLoss` the maximal loss taken over all specified loss distributions

`loc` a locus-function for accessing the list `losses` using a triple notation (goal,i,j), where goal addresses the game matrix and i,j are the row and column indices (starting from 1 as the smallest index). This function is used internally (only).

Warning

Games constructed with real-valued payoffs or payoff vectors over the reals are allowed with negative or zero values in the list of losses. In that case, embeds the loss values or vector into a lossDistribution object after shifting and scaling the values into the strictly positive range. This operation creates a strategically equivalent game, i.e., leaves the set of equilibria unchanged, yet the resulting mosg object *is not* useful with the lossDistribution.mosg, moment, cdf, or any other member functions for lossDistribution objects obtained from equilibria. Those have to be computed manually. Be aware that there will be no warnings issued whatsoever in that case of misuse, since the lossDistribution objects constructed to carry the real or vector-valued payoffs of the original game carry no information about the semantics of the values or vectors that they have been created from. Hence, the *computation of equilibria works correctly* using mosg, while any further analysis (including plots) *needs to be done manually*.

Note

It is important to remark that player 1 is always minimizing. To treat a maximizing player, one must reconstruct the game using regrets instead of losses, i.e., if the data for a specific loss distribution is D, then the game for a maximizing player 1 must be constructed from `(max(D) -D)` instead of `D`.

Author(s)

Stefan Rass

See Also

Security strategies for a `mosg` object can be obtained by calling `mgss`. The game itself can be constructed from the output of `lossDistribution`.

Examples

```
library(compare)

## raw data (PURELY ARTIFICIAL, for demo purposes only)
# N=100 observations in each category
obs111<-c(rep(1,40),rep(3,20),rep(5,10),rep(7,20),rep(9,10));
obs112<-c(rep(1,50),rep(2,10),rep(4,10),rep(6,20),rep(8,10));
obs121<-c(rep(1,20),rep(4,30),rep(6,20),rep(8,10),rep(10,20));
obs122<-c(rep(1,40),rep(2.5,20),rep(5,20),rep(7.5,10),rep(9,10));
obs211<-c(rep(1,30),rep(2,30),rep(5,10),rep(8,10),rep(10,20));
obs212<-c(rep(1,10),rep(2,10),rep(4,20),rep(7,20),rep(10,40));
```

```
obs221<-c(rep(1,30),rep(3,30),rep(4,10),rep(7,20),rep(9,10));
obs222<-c(rep(1,10),rep(3,10),rep(5,50),rep(8,20),rep(10,10));
obs311<-c(rep(1,40),rep(2,30),rep(4,10),rep(7,10),rep(9,10));
obs312<-c(rep(1,20),rep(3,20),rep(4,20),rep(7,20),rep(10,20));
obs321<-c(rep(1,10),rep(3,40),rep(4,30),rep(7,10),rep(9,10));
obs322<-c(rep(1,10),rep(4,30),rep(5,30),rep(7,10),rep(10,20));

## compute payoff densities
f111<-lossDistribution(obs111)
f112<-lossDistribution(obs112)
f121<-lossDistribution(obs121)
f122<-lossDistribution(obs122)
f211<-lossDistribution(obs211)
f212<-lossDistribution(obs212)
f221<-lossDistribution(obs221)
f222<-lossDistribution(obs222)
f311<-lossDistribution(obs311)
f312<-lossDistribution(obs312)
f321<-lossDistribution(obs321)
f322<-lossDistribution(obs322)

payoffs<-list(f111,f112,f121, f122,f211,f212,f221,f222, f311,f312,f321,f322)
G <- mosg( n=2,
           m=2,
           payoffs,
           goals=3,
           goalDescriptions=c("g1", "g2", "g3"),
           defensesDescr = c("d1", "d2"),
           attacksDescr = c("a1", "a2"))
print(G)
summary(G)
plot(G)

# construct and solve scalar valued (classical) game;
# losses are all numbers (degenerate distributions)
# the resulting matrix game has the payoff structure:
#     [,1] [,2]
#[1,]    3    4
#[2,]    6    1
G <- mosg(n = 2, m = 2, goals = 1, losses = list(3,6,4,1), byrow=FALSE)
mgss(G)  # compute an equilibrium
```

`mosg.equilibrium` *embodies all information related to an equilibrium computed by the function* `mgss`.

Description

The generic functions `print` and `summary` provide brief, and detailed information about the equilibrium. The generic function `plot` can be used to visualize the equilibrium.

Usage

```
## S3 method for class 'mosg.equilibrium'
summary(object, ...)
## S3 method for class 'mosg.equilibrium.summary'
print(x, ...)
## S3 method for class 'mosg.equilibrium'
print(x, extended=FALSE, ...)
## S3 method for class 'mosg.equilibrium'
plot(x, points=100, ...)
```

Arguments

`x`	an `mgss` object as returned by the function `mgss`.
`object`	an `mgss` object as returned by the function `mgss`. for `print.mosg.equilibrium`, the following parameter can be supplied:
`extended`	if set to `TRUE`, then the individual assurances are printed as well. for `plot.mosg.equilibrium`, the following parameter can be supplied:
`points`	the number of points to evaluate the density function over its support for plotting
`...`	further arguments passed to or from other methods.

Value

the result returned by the function `summary` carries the following fields:

`optimalDefense`	a discrete probability distribution over the action space for player 1 (the defender).
`optimalAttacks`	a discrete probability distribution over the action space for player 2 (the attacker).
`assurances`	an optimal loss distribution valid under the assumption that the defender plays `optimalDefense` as its mixed strategy. This is a list of `mosg.lossdistribution` objects, accessible through their assigend names (coming from the underlying game) or by indices.

The action spaces for both players are defined in first place by the game for which the equilibrium was computed (via `mgss` on a game constructed by `mosg`).

`print` gives a shortened output restricted only to displaying the optimal defense for the defender and attack strategies per goal (as defined by the underlying game).

`summary` returns an object of class `summary.mosg.lossdistribution`, which has the fields: "range" "mean" "variance" "quantiles" "is.discrete"

`range`	the minimal and maximal values of the loss (as anticipated by the observations)
`mean`	the first moment as computed by `mean`
`variance`	the variance as computed by `variance`
`quantiles`	a 2x5-matrix of quantiles at the 10%,25%,50%,75% and 90% level
`is.discrete`	a Boolean flag being `TRUE` if the loss distribution is over categories

plot displays a grid of plots, starting with the optimal defense behavior plotted as a discrete distribution on top of a (m x 2)-matrix of plots. Each line in this grid shows the discrete optimal attack strategy on the right side (as a bar plot), paired with the loss distribution (extracted from x) caused when the defender plays optimalDefense and the attacker plays the respective optimal attack strategy.

Author(s)

Stefan Rass

See Also

print.mosg.equilibrium, mgss, mosg, lossDistribution

Examples

```
library(compare)
library(orthopolynom)
## raw data (PURELY ARTIFICIAL, for demo purposes only)
# N=100 observations in each category
obs111<-c(rep(1,40),rep(3,20),rep(5,10),rep(7,20),rep(9,10));
obs112<-c(rep(1,50),rep(2,10),rep(4,10),rep(6,20),rep(8,10));
obs121<-c(rep(1,20),rep(4,30),rep(6,20),rep(8,10),rep(10,20));
obs122<-c(rep(1,40),rep(2.5,20),rep(5,20),rep(7.5,10),rep(9,10));
obs211<-c(rep(1,30),rep(2,30),rep(5,10),rep(8,10),rep(10,20));
obs212<-c(rep(1,10),rep(2,10),rep(4,20),rep(7,20),rep(10,40));
obs221<-c(rep(1,30),rep(3,30),rep(4,10),rep(7,20),rep(9,10));
obs222<-c(rep(1,10),rep(3,10),rep(5,50),rep(8,20),rep(10,10));
obs311<-c(rep(1,40),rep(2,30),rep(4,10),rep(7,10),rep(9,10));
obs312<-c(rep(1,20),rep(3,20),rep(4,20),rep(7,20),rep(10,20));
obs321<-c(rep(1,10),rep(3,40),rep(4,30),rep(7,10),rep(9,10));
obs322<-c(rep(1,10),rep(4,30),rep(5,30),rep(7,10),rep(10,20));

## compute payoff densities
f111<-lossDistribution(obs111)
f112<-lossDistribution(obs112)
f121<-lossDistribution(obs121)
f122<-lossDistribution(obs122)
f211<-lossDistribution(obs211)
f212<-lossDistribution(obs212)
f221<-lossDistribution(obs221)
f222<-lossDistribution(obs222)
f311<-lossDistribution(obs311)
f312<-lossDistribution(obs312)
f321<-lossDistribution(obs321)
f322<-lossDistribution(obs322)

payoffs<-list(f111,f112,f121, f122,f211,f212,f221,f222, f311,f312,f321,f322)
G <- mosg( n=2,
            m=2,
            payoffs,
            goals=3,
```

```
            goalDescriptions=c("g1", "g2", "g3"),
            defensesDescr = c("d1", "d2"),
            attacksDescr = c("a1", "a2"))
eq <- mgss(G,weights=c(0.25,0.5,0.25))
print(eq)
summary(eq)
plot(eq)

# access the loss distributions computed in the game
summary(eq$assurances$g1)
mean(eq$assurance$g1)    # get the average loss in goal "g1"
```

`preference` *Decision on preferences between loss distributions*

Description

This function implements the total ordering on losses, based on treating the moment sequences as hyperreal numbers, and returns the lesser of the loss distribution representatives in the hyperreal space.

Usage

```
preference(x, y, verbose = FALSE, weights, points = 512)
```

Arguments

`x`	a loss, being either a number,a distribution or list of distributions (objects of class `mosg.lossdistribution`)
`y`	a loss, being either a number,a distribution or list of distributions (objects of class `mosg.lossdistribution`)
`weights`	a vector of n = `length(x)` = `length(y)` nonzero numbers (not necessarily summing up to 1), used only if `x` and `y` are lists of `mosg.lossdistribution` objects corresponding to $n > 1$ goals. In that case, the i-th goal gets assigend the weight (priority) `weights[[i]]`. Defaults to all goals having equal priority if the parameter is missing (`weights` = `rep(1/length(x),length(x))`).
`verbose`	if set to `TRUE`, the function returns the preferred of its arguments directly (thus, giving back `x` or `y`). If set to `FALSE` (default), then it returns the argument index (1 = `x`, 2 = `y`) or 0 in case that $x = y$.
`points`	the number of points at which the distributions are evaluated numerically to determine the preference.

Details

Deciding the preference ordering defined in terms of moment sequence as proposed in (Rass, 2015). To avoid having to compute all moments up to an unknown order, this function decides by looking at the tails of the distribution, returning the one with faster decaying tail as the preferred distribution. This method delivers exact decisions for discrete distributions, but is only an approximate approach for continous densities.

Value

the result is either a copy of the input parameter x or y, depending on which distribution is preferred.

Author(s)

Stefan Rass

References

S. Rass. On Game-Theoretic Risk Management (Part One). Towards a Theory of Games with Payoffs that are Probability-Distributions. ArXiv e-prints, June 2015. http://arxiv.org/abs/1506.07368.

See Also

`lossDistribution`, `lossDistribution.mosg`, `print.mosg.lossdistribution`

Examples

```
# use data from CVSS risk assessments
cvss1base <- c(10,6.4,9,7.9,7.1,9)
cvss2base <- c(10,7.9,8.2,7.4,10,8.5,9,9,8.7)
ld1 <- lossDistribution(cvss1base)
ld2 <- lossDistribution(cvss2base)
lowerRisk <- preference(ld1, ld2) # get the result for later use
preference(ld1, ld2, verbose=TRUE) # view the detailed answer
```

`variance` *Computes the approximate variance of a loss distribution.*

Description

The computation is based on Steiner's theorem $\mathrm{var}(X) = \mathrm{E}(X^2) - (\mathrm{E}(X))^2$, where the respective first and second moments are computed using the `moment` function (from this package). Internally, these functions operate on the approximate kernel density estimation for both, continuous and categorical distributions (see the `lossDistribution` function for details).

Usage

```
variance(x)
```

Arguments

x	an object of class `mosg.lossDistribution`

Value

the approximate variance value

Note

the function works on the internal probability mass function (which may be different from the empirical distribution in case that the loss distribution has been smoothed during its construction; see lossDistribution). The function delivers only an approximate variance, whose error is due to numeric roundoff errors (known to occur in Steiner's formula), and the fact that the computation is done on an approximate density (rather than the empirical distribution).

Author(s)

Stefan Rass

See Also

moment, lossDistribution

Examples

```
x <- c(10,6.4,9,7.9,7.1,9)
ld <- lossDistribution(x)
variance(ld)
var(x)
```

[.mosg   *Extract or replace parts of a game's payoff matrix*

Description

Construct a new game by taking out a specified set of rows, columns and goals from a given game G. The new game inherits all descriptions (rows, cols and goals) from the G, and has its list of loss distributions organized in the same way (by rows or columns) as G.

The extraction or substitution works like as for data frames (see [.data.frame). Strategies for both players, as well as goals, can equivalently be addressed by their string-names.

Usage

```
## S3 method for class 'mosg'
x[i,j,k=NULL]

## S3 replacement method for class 'mosg'
x[i,j,k=NULL] <- value
```

Arguments

`x`	a game of class mosg
`i,j,k`	a numeric value or numeric vector of row incides `i`, colum indices `j`, or goals `k`.
`value`	a list of lossDistribution objects, or a game object of class `mosg`.

Details

For `[` extraction of elements from a payoff matrix, omitting any index dimension selects all elements in the respective dimension. Supplying negative values excludes the respective elements. For example, `G[c(1:3),1]` returns a game with only the rows 1..3 of `G`, but all column strategies that `G` had, and only the first of `G`'s goals retained.

For `[<-`, the list of substitute values needs to be of the same length as the number of elements addressed by the triple `(i,j,k)`, otherwise an error is returned. If the new elements come from another game object, say `G2`, only the loss distributions get replaced, but not the names of the strategies. The replacement checks if `G2` has its list of loss distributions organized in the same way as `G`, i.e., row-by-row or column-by-column. If there is a mismatch, the substitution is nonetheless done, but a warning about this issue is printed.

Value

`[` returns a freshly constructed game object.

Warning

For `[<-`, be aware that the replacement *does not* semantically check if the newly incoming loss distributions make sense as elements of the new game (e.g., they can have different supports, or be discrete/continuous while the game was continuous/discrete in its payoffs). Respective errors may only subsequently come up when the modified or extracted game is used.

Author(s)

Stefan Rass

See Also

`[.data.frame`

Examples

```
## raw data (PURELY ARTIFICIAL, for demo purposes only)
obs111<-c(rep(1,40),rep(3,20),rep(5,10),rep(7,20),rep(9,10));
obs112<-c(rep(1,50),rep(2,10),rep(4,10),rep(6,20),rep(8,10));
obs121<-c(rep(1,20),rep(4,30),rep(6,20),rep(8,10),rep(10,20));
obs122<-c(rep(1,40),rep(2.5,20),rep(5,20),rep(7.5,10),rep(9,10));
obs211<-c(rep(1,30),rep(2,30),rep(5,10),rep(8,10),rep(10,20));
obs212<-c(rep(1,10),rep(2,10),rep(4,20),rep(7,20),rep(10,40));
obs221<-c(rep(1,30),rep(3,30),rep(4,10),rep(7,20),rep(9,10));
obs222<-c(rep(1,10),rep(3,10),rep(5,50),rep(8,20),rep(10,10));
obs311<-c(rep(1,40),rep(2,30),rep(4,10),rep(7,10),rep(9,10));
obs312<-c(rep(1,20),rep(3,20),rep(4,20),rep(7,20),rep(10,20));
obs321<-c(rep(1,10),rep(3,40),rep(4,30),rep(7,10),rep(9,10));
obs322<-c(rep(1,10),rep(4,30),rep(5,30),rep(7,10),rep(10,20));
## compute payoff densities
f111<-lossDistribution(obs111)
f112<-lossDistribution(obs112)
f121<-lossDistribution(obs121)
f122<-lossDistribution(obs122)
f211<-lossDistribution(obs211)
f212<-lossDistribution(obs212)
f221<-lossDistribution(obs221)
f222<-lossDistribution(obs222)
f311<-lossDistribution(obs311)
f312<-lossDistribution(obs312)
f321<-lossDistribution(obs321)
f322<-lossDistribution(obs322)

payoffs<-list(f111,f112,f121, f122,f211,f212,f221,f222, f311,f312,f321,f322)
G <- mosg( n=2,
            m=3,
            payoffs,
            goals=2,
            goalDescriptions=c("g1", "g2"),
            defensesDescr = c("d1", "d2"),
            attacksDescr = c("a1", "a2", "a3"))

# modify the game by subsetting
G[,c(1,2),] # select only the first two strategies
G[,-3,] # exclude the third strategy (equivalent to before)

# replace a 2x2 subgame related to the second goal
# (replacement data is chosen arbitrarily here)
G2 <- mosg(n=2, m=2, goals=1, losses = list(f111,f112,f121, f122))
G[,c(1,2),1] <- G2  # replace the subgame

# construct another replacement game that is organized different (by column)
G2 <- mosg(n=2, m=2, goals=1, losses = list(f111,f112,f121, f122), byrow=FALSE)
G[,c(1,2),1] <- G2  # this will issue a warning

# plot a submatrix from the game
plot(G[-2,c(1,2),], goal=2)
```

Index