# Package 'frscore'

April 28, 2023

**Title** Functions for Calculating Fit-Robustness of CNA-Solutions

**Version** 0.3.1

**Description** Functions for automatically performing
a reanalysis series
on a data set using CNA, and for calculating the fit-robustness
of the resulting models, as described in
Parkkinen and Baumgartner (2021) <doi:10.1177/0049124120986200>.

**License** AGPL (>= 3)

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 7.2.3

**Imports** dplyr, Rfast, magrittr, rlang

**Depends** R (>= 3.5), cna (>= 3.5.1), lifecycle (>= 1.0.0)

**Suggests** spelling, testthat (>= 3.0.0)

**Config/testthat/edition** 3

**Language** en-US

**NeedsCompilation** no

**Author** Veli-Pekka Parkkinen [aut, cre, cph],
Michael Baumgartner [aut, cph],
Mathias Ambuehl [aut, cph]

**Maintainer** Veli-Pekka Parkkinen <parkkinenv@gmail.com>

**Repository** CRAN

**Date/Publication** 2023-04-28 13:50:02 UTC

## R topics documented:

causal_submodel . . . . . 2
d.error . . . . . 6
frscore . . . . . 6
frscored_cna . . . . . 9
rean_cna . . . . . 12

**Index** **15**

---

`causal_submodel` *Determine if a model is a causal submodel of another model*

---

## Description

Determine whether the causal relevance ascriptions made by *candidate* solution/model `x` are contained in the causal relevance ascriptions made by *target* model `y`.

## Usage

```
causal_submodel(x, y, dat = NULL)
```

## Arguments

| | |
|---|---|
| `x` | A string that specifies a valid `cna` model. |
| `y` | A string that specifies a valid `cna` model. |
| `dat` | A `configTable`, a data frame, a matrix, or a list that specifies the range of admissible factor values for the factors featured in `x` and `y`. Only needed when the models `x` and `y` are multi-valued, otherwise ignored. |

## Details

`causal_submodel()` checks whether the causal relevance claims made by the candidate model `x` are contained within the causal relevance claims made by the target model `y`. When `x` and `y` are multi-valued models, a further argument `dat` must be provided to determine the admissible factor values for the factors featured in `x` and `y`. This would typically be the data set that `x` and `y` were inferred from. `causal_submodel()` is similar to, and based on `is.submodel()` from the `cna` package, with one important difference. `is.submodel()` checks whether a model is a syntactic submodel of another, and can thus be used to check whether all syntactically explicit causal ascriptions, i.e. claims about direct causation only, of one model are contained in another. `causal_submodel()` checks if *all* causal *relevance* claims made by `x`, i.e. claims of either direct or indirect causation, have a counterpart causal relevance ascription in `y`. In case when all causal relevance claims of `x` have a suitable (see below) counterpart in `y`, `x` is a *causal* submodel of `y`.

For `x` to be causal submodel of `y`, (1), every ascription of direct causal relevance made by `x` must either have a counterpart direct causal ascription in `y`, or a counterpart indirect causal ascription in `y` such that `x` omits any factors that mediate the relation according to `y`. (2), every ascription of indirect causal relevance made by `x` must have a counterpart indirect causal ascription in `y`. That is, every pair of factors represented as direct cause and effect in `x` must either be represented as direct cause and effect in `y`, or be connected by a transitive chain of direct causal relations according to `y`. In the latter case, `x` must in addition omit the factors that according to `y` mediate the causal relation in question. Direct causal relations are those causal relations that can be read off from the explicit syntax of an atomic solution/model ("asf"). For example, according to `A*F+B<->C`, `A` and `B` are direct causes of `C` on alternative paths. Furthermore, candidate model `A+B<->C` is a causal submodel of the target `A*F+B<->C`, but `A+B*U<->C` is not, since the latter makes a claim about the causal relevance of `U` to `C` which is not made by the target. Each direct cause is a difference-maker

for its effect in some circumstances where alternative sufficient causes of the effect are not present, and the *co-factors* located on the same path are present. For example, `A*F+B<->C` claims that when `B` is absent and `F` is present, difference in the presence of `A` will associate with differences in `C`, given some suitable configuration of factors not explicitly represented in `A*F+B<->C`. When both `x` and `y` are asfs, i.e. represent direct causal relations only, `x` is a causal submodel of `y` if, and only if `x` is is a syntactic submodel of `y`, as the syntax of an asf is such that every causal ascription is explicitly represented.

Judgments of direct vs. indirect causation are relative to the set of factors included in a model. `A+B<->E` describes `A` and `B` as direct causes of `E`, but another model that includes additional factors besides `{A,B,E}` might describe these causal relations as causal chains that include intermediate steps, as in `(A+B<->C)*(C+D<->E)`. `A+B<->E` makes no claim that would contradict the chain model; it merely says that *relative* to the factor set `{A,B,E}`, the factors are causally ordered so that `A` and `B` are causes of `E`, and there is no causal relation between `A` and `B`. Causal order refers to the ordering of the factors by the relation of direct causation that determines what is causally "upstream" and "downstream" of what. The exogenous factors `{A,B,D}` are top-level upstream causes in `(A+B<->C)*(C+D<->E)`, as they are not caused by any other factor included in the model. Endogenous factors `C` and `E` are downstream of of `{A,B}` by one and two levels respectively, and `E` is one level downstream of `D`. The chain model agrees with the direct cause model on the causal ordering of `{A,B,E}` – `A` and `B` are upstream of `E` and not causes of each other – but also includes an additional cause of `E`, `C`, that is ordered between `{A,B}` and `E` along a chain of direct causal relations. `(A+B<->C)*(C+D<->E)` represents a *transitive* causal chain where `A` and `B` are indirectly causally relevant for `E` in virtue of being causes of `E`'s more proximate cause `C` and the difference-making ability they have on `E` via `C`. `A+B<->E` is a causal submodel of `(A+B<->C)*(C+D<->E)`, as the models agree on the causal relevance ascriptions over `{A,B,E}`, and the former makes no claims whatsoever about the additional factors `{C,D}` included in the latter model. Both models can be seen as descriptions of the same causal structure, one more complete in detail than the other. An *in*transitive chain is a causal chain where the influence of some upstream causes is not transmitted to some downstream effects. For example, `(A+B<->C)*(C*a+D<->E)` represents a chain where `A` is not causally relevant to `E` despite being a cause of one of `E`'s direct causes (`C`). That is, according to this model, `A` is not a difference-maker for `E`, and `A+B<->E`, which makes this claim, is not its causal submodel.

Besides avoiding causal relevance ascriptions that are not present in the target at all, the candidate should also attribute causal relevance correctly in the sense of causally ordering the represented causes in a way that is compatible with the target. Factors that appear as direct causes of the same outcome both in the target and the candidate should be grouped into alternative disjuncts similarly in both. Analogously, causes that appear on different levels in a causal chain according to the target should not be represented as same-level causes by the candidate. Say, for example, that the target is `(A+B<->C)*(C+D<->E)*(E+F<->G)`. Candidate models `(A+B<->G)` and `(A+B<->E)*(E+F<->G)` are both causal submodels of this target. By contrast, neither of `(A+C<->G)` and `(A+B<->C)*(C+E<->G)` is a causal submodel of the target. Both of the latter two models commit the error of representing as same-level causes factors that the target represents as cause and effect. For example, `(A+C<->G)` claims that `A` and `C` are same-level causes of `G`, whereas the target says `A` is a cause of `C`. In other words, relative to a factor set that include `C`, the candidate claims that `A` is a direct cause of `E`, which is false according to the target. It is instructive to consider the difference in implications for difference-making: `(A+C<->G)` claims that differences in `A` associate with differences in `G` when `C` is fixed absent, but the target claims that this is impossible.

Finally, a causal submodel relation requires that any claims of indirect causal relevance made by a candidate model are claims made by the target also. Consider the target model `(A+B*D<->C)*(C+D<->G)`

and a candidate `(A+B*D<->C)*(C<->G)`. Despite superficial similarity (the candidate is a syntactic submodel of the target), the candidate is not a causal submodel of the target. Namely, the candidate makes a claim that `B` is indirectly causally relevant for `G`, a claim that is not made by the target. Again, it is best to examine the specific difference-making claim in question. The candidate model claims that differences in `B` make a difference to the presence of `G` when `D` is fixed to be present. But this is false according to the target. The target claims that `G` is always present whenever `D` is: `B` is not causally relevant for `G` despite being a cause of an intermediary factor `C`.

In its implementation, `causal_submodel()` relies on the fact that when both the target and candidate are asfs, a syntactic submodel relation that can be verified with `is.submodel()` is a necessary and sufficient condition for causal submodel relation. If both the candidate and the target are asfs, a check for syntactic submodel relation is performed, and the result returned. When the target, or both the target and candidate comprise more than one asf, the process is more complicated. First, `causal_submodel()` checks if the component asfs of the candidate are syntactic submodels of the target *as is*. If yes for all, each of the candidate's direct causal relevance ascriptions is contained in the target, and the function proceeds to the second phase. For those direct causal relations that are not contained in the target, the function searches for counterpart indirect relations in the target. Since `cna` models do not represent indirect relations explicitly, these are explicated by syntactically manipulating the target. This involves finding asfs in the target with the same outcomes as those candidate asfs that are not syntactic submodels of the target *as is*. For each such component asf of the target, factors in the disjunction on the left hand side of the equivalence sign ("<->") are substituted with the disjunctions, if any, that according to the target represent their causes. The resulting expression is then minimized to render it causally interpretable. What is left is an asf representing some of the target's indirect causal claims as direct causal claims. Then, the candidate asfs that are not syntactic submodels of the target *as is* are tested against the manipulated target asfs for syntactic submodel relation. This process is repeated until all the submodel checks return `TRUE`, or no further substitutions are possible. In the former case, the function proceeds to the second phase. In the latter case, the candidate is deemed not to be a causal submodel of the target, and the function returns `FALSE`.

An example is in order to illustrate the procedure so far. Say that the target and candidate are `(A+B<->C)*(C+D<->E)` and `A+B<->E`, respectively. Since the sole candidate asf is not a syntactic submodel of the target, one then attempts to find indirect causal relevance ascriptions in the target to license the direct causal claims made by the candidate asf. By the procedure described above, one focuses on the second asf of the target, `C+D<->E`, and seeks to syntactically manipulate that until it is transformed into a syntactic supermodel of `A+B<->E`, or until no transformation is possible. According to the first component asf of the target, `C` is equivalent to (caused by) `A+B`. Hence, `C` in `C+D<->E` can be replaced with `A+B`, which yields `(A+B)+D<->E`, reducing simply to `A+B+D<->E`. Since `A+B<->E` is a syntactic submodel of `A+B+D<->E`, we have shown that the causal relevance claims made by the candidate are contained in the target.

The purpose of the second phase is to check that all indirect causal claims made by the *candidate* model have a counterpart in the target. This involves doing all the substitutions of left-hand side factors by their causes in the candidate model, to generate expressions that explicitly represent the indirect claims of the candidate. The asfs generated by such manipulations of the candidate model are then tested for causal compatibility with the target, following the exact same procedure described above. For example, say that `(A+B*D<->C)*(C+D<->G)` and `(A+B*D<->C)*(C<->G)` are the target and the candidate, respectively. Here, each candidate asf `A+B*D<->C` and `C<->G` has a supermodel in one of the target asfs `A+B*D<->C` and `C+D<->G`, i.e. each direct causal claim of the candidate has a counterpart direct causal claim in the target, and the function proceeds to the second phase. In the second phase, the indirect causal claims of the candidate are first made explicit.

By substituting A+B*D in place of C in the second asf of the candidate and minimizing, one gets A+B*D<->G, which represents the indirect causal relevance, as claimed by the candidate, of A+B*D on G. This expression is then tested against the target as in the first phase: the target asf with G as the outcome is manipulated to reflect the indirect claims that the target makes about G, based on what the target says about the indirect causes of G. After substitution and minimization, we get A+D<->G, meaning that the target does *not* make a claim of indirect causal relevance of B for G. That the candidate's indirect causal ascriptions are not contained in the target is shown by the fact that A+B*D<->G is not a syntactic submodel of A+D<->G, and the function returns FALSE.

Due to the computational demands of some of the steps in the above procedure, causal_submodel() is an approximation of a strictly speaking valid check for causal submodel relations. Since the syntactic manipulations and especially the minimization of the resulting expressions is so costly, causal_submodel() relies on the rreduce() function from the cna package for minimization. rreduce() randomly chooses a single reduction path to produce only one minimal form of an expression whenever more than one exists, i.e. when the expression is ambiguous in its causal claims. In the case of ambiguous models, the output of causal_submodel() may depend on which reduction path(s) were chosen. These cases are rare enough to not significantly affect the intended use of causal_submodel() in the context of frscore. Another instance of causal_submodel() taking a shortcut is when processing cyclic models like (A+B<->C)*(C+D<->A). Here the problems are as much philosophical as computational. It is clear that a cyclic candidate model cannot be a causal submodel of a non-cyclic target. However, problems arise when testing a non-cyclic candidate against a cyclic target: it is not clear what counts as an incompatibility in causal ordering, given that a cyclic target model includes factors that are causally relevant for themselves. Since many conclusions can be argued for here but some approach must be taken to ensure that causal_submodel() works on all valid cna models, causal_submodel() takes the least costly option and simply checks whether the candidate is a syntactic submodel of the target, and returns the result.

## Value

Named logical.

## See Also

cna::is.submodel()

## Examples

```
target <- "(A+B<->C)*(C+D<->E)"
candidate1 <- "A+B<->E"
causal_submodel(candidate1, target) # TRUE
candidate2 <- "A+C<->E"
causal_submodel(candidate2, target) # FALSE

dat <- cna::d.pban
target_mv <- "C=1 + F=2 + T=1 + C=0*F=1 <-> PB=1"
candidate_mv <- "C=1 + F=2 + T=1 <-> PB=1"
causal_submodel(candidate_mv, target_mv, dat = dat) # mv models require the 'dat' argument
```

## d.error *Simulated data of sixteen cases with measurement error in one case*

### Description

A simulated set of crisp-set configurational data that conforms to the structure A + B*C <-> E except for one case, 'c16', which simulates measurement error in that case, and including one irrelevant factor "D".

### Usage

```
d.error
```

### Format

An object of class `data.frame` with 16 rows and 5 columns.

## frscore *frscore*

### Description

Calculate fit-robustness scores for a set of `cna` solutions/models

### Usage

```
frscore(
  sols,
  dat = NULL,
  scoretype = c("full", "supermodel", "submodel"),
  normalize = c("truemax", "idealmax", "none"),
  maxsols = 50,
  verbose = FALSE,
  print.all = FALSE,
  comp.method = c("causal_submodel", "is.submodel")
)
```

### Arguments

| | |
|---|---|
| `sols` | Character vector of class "stdAtomic" or "stdComplex" (as generated by `cna()`) that contains the solutions/models to be scored. |
| `dat` | A `configTable`, a data frame, a matrix, or a list that specifies the range of admissible factor values for the factors featured in the models included in `sols`. Only needed when the models in `sols` are multi-valued, otherwise ignored. |

`scoretype` **[Deprecated]** String specifying the scoring method: `"full"` (default; scoring is based on counting sub- and supermodel relations), `"supermodel"` (count supermodels only), `"submodel"` (count submodels only). Allowed for backward compatibility only, due to be dropped in next version.

`normalize` String that determines the method used in normalizing the scores. `"truemax"` (default) normalizes by the highest score among the elements of `sols`, such that the highest scoring solution types get score 1. `"idealmax"` normalizes by a theoretical maximum score (see Details).

`maxsols` Integer determining the maximum number of unique solution types found in `sols` to be included in the scoring (see Details).

`verbose` Logical; if `TRUE`, additional information about causal compatibility relations among the unique solution types found in `sols` is printed. Defaults to `FALSE`.

`print.all` Logical, controls the number of entries printed when printing the results. If `TRUE`, results are printed as when using the defaults of `print.data.frame`. If `FALSE`, 20 highest scoring solutions/models are printed.

`comp.method` String that determines how the models in `sols` are compared to determine their fr-score. `"causal_submodel"` (the default) checks for causal submodel relations using `causal_submodel()`, `"is.submodel"` checks for syntactic submodel relations with `is.submodel()`

## Details

`frscore()` implements fit-robustness scoring as introduced in Parkkinen and Baumgartner (2021). The function calculates the fit-robustness scores of Boolean solutions/models output by the `cna()` function of the **cna** package. The solutions are given to `frscore()` as a character vector `sols` obtained by reanalyzing a data set repeatedly, e.g. with `rean_cna()`, using different consistency and coverage thresholds in each analysis.

For multi-valued models, the range of admissible values for the factors featured in the models must be provided via the argument `dat`, which accepts a data frame, `configTable`, or a list of factor-value ranges as its value, in the same manner as `cna::full.ct()`. Typically, one would use the data set that the models in `sols` were inferred from, and this is what is done automatically when `frscore()` is called within `frscored_cna()`. When the models in `sols` are binary, `dat` should be left to its default value `NULL`, and will in any case be ignored.

The argument `scoretype` is deprecated as of `frscore` 0.3.1, and will be dropped in the next version. Giving it a non-default value is allowed so that older code can be run without errors, but doing this is otherwise discouraged. When set to its default value `"full"`, the score of each `sols[i]` is calculated by counting the (either syntactic or causal) sub- and supermodel relations `sols[i]` has to the other elements of `sols`. Setting scoretype to `"supermodel"` or `"submodel"` forces the scoring to be based on, respectively, supermodel and submodel relations only. Whether causal or syntactic submodel relations are counted depends on the value of `comp.method`: `"causal_submodel"` (default) counts causal submodel relations using `causal_submodel()`, `"is.submodel"` counts syntactic submodel relations using `cna::is.submodel()`. In future versions of `frscore`, fit-robustness scores will always be calculated as with `scoretype = "full"`, and changing this will not be possible. If additional information about the numbers of sub- vs. supermodel relations a particular model has to other models is needed, this can be acquired by inspecting the `"verbout"` element of the output of `frscore()`.

The fit-robustness scores can be normalized in two ways. In the default setting `normalize = "truemax"`, the score of each `sols[i]` is divided by the maximum score obtained by an element of `sols`. In case of `normalize = "idealmax"`, the score is normalized not by an actually obtained maximum but by an idealized maximum, which is calculated by assuming that all solutions of equal complexity in `sols` are identical and that for every `sols[i]` of a given complexity, all less complex elements of `sols` are its submodels and all more complex elements of `sols` are its supermodels. When normalization is applied, the normalized score is shown in its own column `norm.score` in the results. The raw scores are shown in the column `score`.

If the size of the consistency and coverage interval scanned in the reanalysis series generating `sols` is large or there are many model ambiguities, `sols` may contain so many different types of solutions/models that robustness cannot be calculated for all of them in reasonable time. In that case, the argument `maxsols` allows for capping the number of solution types to be included in the scoring (defaults to 50). `frscore()` then selects the most frequent solutions/models in `sols` of each complexity level until `maxsols` is reached and only scores the thus selected elements of `sols`.

If the argument `verbose` is set to `TRUE`, `frscore()` also prints a list indicating for each `sols[i]` how many raw score points it receives from which elements of `sols`. The verbose list is ordered with decreasing fit robustness scores.

## Value

A named list where the first element is a data frame containing the unique solution/model types and their scores. Rest of the elements contain additional information about the submodel relations among the unique solutions types and about how the function was called.

## References

V.P. Parkkinen and M. Baumgartner (2021), "Robustness and Model Selection in Configurational Causal Modeling," *Sociological Methods and Research*, doi:10.1177/0049124120986200.

Basurto, Xavier. 2013. "Linking Multi-Level Governance to Local Common-Pool Resource Theory using Fuzzy-Set Qualitative Comparative Analysis: Insights from Twenty Years of Biodiversity Conservation in Costa Rica." *Global Environmental Change* **23** (3):573-87.

## See Also

`rean_cna()`, `cna()`, `causal_submodel()`, `is.submodel()`

## Examples

```
# Artificial data from Parkkinen and Baumgartner (2021)
sols1 <- rean_cna(d.error, attempt = seq(1, 0.8, -0.1))
sols1 <- do.call(rbind, sols1)
frscore(sols1$condition)


# Real fuzzy-set data from Basurto (2013)
sols2 <- rean_cna(d.autonomy, type="fs", ordering = list("EM", "SP"),
         strict = TRUE, maxstep = c(3,3,9))
sols2 <- do.call(rbind, sols2)$condition  # there are 217 solutions
# At the default maxsols only 50 of those solutions are scored.
```

```
frscore(sols2)
# By increasing maxsols the number of solutions to be scored can be controlled.
frscore(sols2, maxsols = 100)


# Multi-valued data/models (data from Hartmann and Kemmerzell (2010))
# Short reanalysis series, change `attempt` value to mimick a more realistic use case
sols3 <- rean_cna(d.pban, outcome = "PB=1", attempt = seq(0.8, 0.7, -0.1), type = "mv")
sols3 <- do.call(rbind, sols3)$condition
# For mv data, frscore() needs the data to determine admissible factor values
frscore(sols3, dat = d.pban)

# Changing the normalization
frscore(sols2, normalize = "none")
frscore(sols2, normalize = "truemax")
frscore(sols2, normalize = "idealmax")

# verbose
frscore(sols2, maxsols = 20, verbose = TRUE)
```

---

`frscored_cna` *frscored_cna*

---

## Description

Perform a reanalysis series on a data set and calculate the fit-robustness scores of the resulting solutions/models

## Usage

```
frscored_cna(
  x,
  fit.range = c(1, 0.7),
  granularity = 0.1,
  output = c("csf", "asf"),
  scoretype = c("full", "supermodel", "submodel"),
  normalize = c("truemax", "idealmax", "none"),
  verbose = FALSE,
  maxsols = 50,
  test.model = NULL,
  print.all = FALSE,
  comp.method = c("causal_submodel", "is.submodel"),
  n.init = 1000,
  ...
)
```

## Arguments

| | |
|---|---|
| x | A data.frame or configTable to be analyzed with cna(). In case of multi-value or fuzzy-set data, the data type must be indicated by type = "mv" and type = "fs", respectively. |
| fit.range | Numeric vector of length 2; determines the maximum and minimum values of the interval of consistency and coverage thresholds used in the reanalysis series. Defaults to c(1, 0.7). |
| granularity | Numeric scalar; consistency and coverage are varied by this value in the reanalysis series. Defaults to 0.1. |
| output | String that determines whether csfs or asfs are returned; "csf" (default) returns csfs, "asf" asfs. |
| scoretype | **[Deprecated]** String specifying the scoring method: "full" (default; scoring is based on counting sub- and supermodel relations), "supermodel" (count supermodels only), "submodel" (count submodels only). Allowed for backward compatibility only, due to be dropped in next version. |
| normalize | String that determines the method used in normalizing the scores. "truemax" (default) normalizes by the highest score among the elements of sols, such that the highest scoring solution types get score 1. "idealmax" normalizes by a theoretical maximum score (see Details). |
| verbose | Logical; if TRUE, additional information about causal compatibility relations among the unique solution types found in sols is printed. Defaults to FALSE. |
| maxsols | Integer determining the maximum number of unique solution types found in the reanalysis series to be included in the scoring (see Details). |
| test.model | String that specifies a single candidate cna() solution/model whose fit-robustness score is calculated against the results of the reanalysis series. |
| print.all | Logical that controls the number of entries printed when printing the results. If TRUE, results are printed as when using the defaults of print.data.frame. If FALSE, 20 highest scoring solutions/models are printed. |
| comp.method | String that determines how the models in sols are compared to determine their fr-score. "causal_submodel" (the default) checks for causal submodel relations using causal_submodel(), "is.submodel" checks for syntactic submodel relations with is.submodel() |
| n.init | Integer that determines the maximum number of csfs built in the analyses, see cna::csf(). Only applied when output = "csf". |
| ... | Any arguments to be passed to cna() except con, cov or con.msc. The effect of argument what is overriden by output. |

## Details

frscored_cna() is a wrapper function that sequentially executes rean_cna() and frscore(), meaning it performs both computational phases of fit-robustness scoring as introduced in Parkkinen and Baumgartner (2021). In the first phase, the function conducts a reanalysis series on the input data x at all combinatorially possible combinations of fit thresholds that can be generated from the interval given by fit.range at increments given by granularity and collects all solutions/models in a set **M**. In the second phase, it calculates the fit-robustness scores of the atomic (asf) and/or

complex (csf) solution formulas in **M**. The argument `output` allows for controlling whether csf or only asf are built, the latter normally being faster but less complete.

The argument `scoretype` is deprecated as of `frscore` 0.3.1, and will be dropped from future versions of the package. Giving it a non-default value is allowed so that older code can be run without errors, but doing this is otherwise discouraged. The permissible values of `scoretype` have the following effects. When set to its default value `"full"`, the score of each solution/model **m** in **M** is calculated by counting the number of the (either causal or syntactic) sub- and supermodel relations **m** has to the other elements of **M**. Whether causal or syntactic submodel relations are counted depends on the value of `comp.method`: `"causal_submodel"` (default) counts causal submodel relations using `causal_submodel()`, `"is.submodel"` counts syntactic submodel relations using `cna::is.submodel()`. Setting `scoretype` to `"supermodel"` or `"submodel"` forces the scoring to be based on, respectively, supermodel and submodel relations only. In future versions of `frscore`, fit-robustness scores will always be calculated as with `scoretype = "full"`, and changing this will not be possible. If additional information about the numbers of sub- vs. supermodel relations a particular model has to other models is needed, this can be acquired by inspecting the `"verbout"` element of the output of `frscored_cna()`.

The fit-robustness scores can be normalized in two ways. In the default setting `normalize = "truemax"`, the score of each `sols[i]` is divided by the maximum score obtained by an element of `sols`. In case of `normalize = "idealmax"`, the score is normalized not by an actually obtained maximum but by an idealized maximum, which is calculated by assuming that all solutions of equal complexity in `sols` are identical and that for every `sols[i]` of a given complexity, all less complex elements of `sols` are its submodels and all more complex elements of `sols` are its supermodels. When normalization is applied, the normalized score is shown in its own column `norm.score` in the results. The raw scores are shown in the column `score`.

If the argument `verbose` is set to `TRUE`, `frscored_cna()` also prints a list indicating for each solution/model how many raw score points it receives from which elements of **M**. The verbose list is ordered with decreasing fit robustness scores.

If the size of the consistency and coverage range scanned in the reanalysis series generating **M** is large or there are many model ambiguities, **M** may contain so many different types of solutions that robustness cannot be calculated for all of them in reasonable time. In that case, the argument `maxsols` allows for capping the number of solution types to be included in the scoring (defaults to 50). `frscored_cna()` then selects the most frequent solutions in **M** of each complexity level until `maxsols` is reached and only scores the thus selected elements of **M**.

If the user is interested in the robustness of one specific candidate model, that model can be given to `frscored_cna()` by the argument `test.model`. The result for that model will then be printed separately, provided the model is found in the reanalysis series, if not, the function stops.

## Value

A list whose first element is a data frame that contains the model types returned from a reanalysis series of the input data, their details such as consistency and coverage, together with the unadjusted fit-robustness score of each model type shown in column 'score', and a normalized score in column 'norm.score' in case `normalize = "truemax"` or `normalize = "idealmax"`. The other elements contain additional information about the submodel relations among the unique solution types and about how the function was called.

## References

P. Emmenegger (2011) "Job Security Regulations in Western Democracies: A Fuzzy Set Analysis." *European Journal of Political Research* 50(3):336-64.

C. Hartmann and J. Kemmerzell (2010) "Understanding Variations in Party Bans in Africa." *Democratization* 17(4):642-65. doi:10.1080/13510347.2010.491189.

V.P. Parkkinen and M. Baumgartner (2021), "Robustness and Model Selection in Configurational Causal Modeling," *Sociological Methods and Research*, doi:10.1177/0049124120986200.

## See Also

`frscore()`, `rean_cna()`, `causal_submodel()`, `cna::is.submodel()`

## Examples

```
# Robustness analysis from sect. 4 of Parkkinen and Baumgartner (2021)
frscored_cna(d.error, fit.range = c(1, 0.75), granularity = 0.05,
             ordering = list("E"), strict = TRUE)

# Multi-value data from Hartmann and Kemmerzell (2010)
frscored_cna(d.pban, type = "mv", fit.range = c(0.9, 0.7), granularity = 0.1,
                  normalize = "none", ordering = list("T", "PB"), strict = TRUE)

# Fuzzy-set data from Emmenegger (2011)
frscored_cna(d.jobsecurity, type = "fs", fit.range = c(0.9, 0.6), granularity = 0.05,
                  scoretype = "submodel", ordering = list("JSR"), strict = TRUE)

# Artificial data
dat <- data.frame(
  A = c(1,1,0,0,0,0,1,1),
  B = c(0,1,0,0,0,0,1,1),
  C = c(1,0,1,0,1,0,1,0),
  D = c(1,1,0,0,1,1,0,0),
  E = c(1,1,1,1,0,0,0,0))
frscored_cna(dat)
frscored_cna(dat, output = "asf")
frscored_cna(dat, maxsols = 10)
frscored_cna(dat, test.model = "(b*e+A*E<->D)*(B<->A)")
```

---

`rean_cna` *rean_cna*

---

## Description

Perform a reanalysis series on a data set with `cna()` using all combinations of consistency and coverage threshold values in a given range of values

## Usage

```
rean_cna(x, attempt = seq(1, 0.7, -0.1), ncsf = deprecated(), output = c("csf", "asf"),
n.init = 1000, ...)
```

## Arguments

| | |
|---|---|
| x | A data.frame or configTable to be analyzed with cna(). In case of multi-value or fuzzy-set data, the data type must be indicated by type = "mv" and type = "fs", respectively. |
| attempt | Numeric vector that contains the values from which combinations of consistency and coverage thresholds are formed, to be used in the analyses. |
| ncsf | **[Deprecated]** Allowed for backward compatibility, due to be dropped in future versions. Please use n.init instead. |
| output | Character vector that determines whether csfs or asfs are returned; "csf" (default) returns csfs, "asf" asfs. |
| n.init | Integer that determines the maximum number of csfs built in the analyses. See csf() |
| ... | Any arguments to be passed to cna() except con, cov or con.msc. The effect of argument what is overriden by output. |

## Details

rean_cna() performs a reanalysis series of a data set x, which constitutes the first computational phase of fit-robustness scoring as introduced in Parkkinen and Baumgartner (2021). The series consists of cna() calls at all combinatorially possible consistency and coverage settings drawn from the vector attempt. If the output argument is set to its default value "csf", rean_cna() returns complex solutions formulas (csf), in case of "asf" only atomic solution formulas ("asf") are built, which is faster. The argument n.init allows for controlling the number of csf to be built, if output = "csf".

## Value

A list where each element is a data frame containing the results of a single analysis of the input data set with cna(), each using a different combination of consistency and coverage threshold values. These values are added to the output as extra columns 'cnacon' and 'cnacov'.

## References

V.P. Parkkinen and M. Baumgartner (2021), "Robustness and Model Selection in Configurational Causal Modeling," *Sociological Methods and Research*, doi:10.1177/0049124120986200.

## See Also

frscore(), cna()

## Examples

```
# Crisp-set data
sols1 <- rean_cna(d.error, attempt = seq(1, 0.8, -0.1))
sols1 <- do.call(rbind, sols1)
sols1

# Multi-value data
sols2 <- rean_cna(d.pban, type = "mv", attempt = seq(0.9, 0.7, -0.1),
                  ordering = list("T", "PB"), strict = TRUE)
sols2 <- do.call(rbind, sols2)
sols2

# Fuzzy-set data
sols3 <- rean_cna(d.jobsecurity, type = "fs", attempt = seq(0.9, 0.7, -0.1),
               ordering = list("JSR"), strict = TRUE) # execution takes a couple of seconds
sols3 <- do.call(rbind, sols2)
sols3
```

# Index

∗ **datasets**
d.error, 6

causal_submodel, 2
causal_submodel(), *7, 8*, *11, 12*
cna, *2*, *5*
cna(), *6–8*, *10*, *12, 13*
cna::csf(), *10*
cna::full.ct(), *7*
cna::is.submodel(), *5*, *7*, *11, 12*
csf(), *13*

d.error, 6

frscore, 6
frscore(), *10*, *12, 13*
frscored_cna, 9

is.submodel(), *2*, *4*, *8*

rean_cna, 12
rean_cna(), *8*, *10*, *12*
rreduce(), *5*

syntactic submodel, *3*