Package 'cbcTools'

May 8, 2026

Title Design and Analyze Choice-Based Conjoint Experiments

Version 0.7.1

Maintainer John Helveston <john.helveston@gmail.com>

Description

Design and evaluate choice-based conjoint survey experiments. Generate a variety of survey designs, including random designs, frequency-based designs, and D-optimal designs, as well as ``labeled'' designs (also known as ``alternative-specific designs''), designs with ``no choice'' options, and designs with dominant alternatives removed. Conveniently inspect and compare designs using a variety of metrics, including design balance, overlap, and D-error, and simulate choice data for a survey design either randomly or according to a utility model defined by user-provided prior parameters. Conduct a power analysis for a given survey design by estimating the same model on different subsets of the data to simulate different sample sizes. Bayesian D-efficient designs using the 'cea' and 'modfed' methods are obtained using the 'idefix' package by Traets et al (2020) <doi:10.18637/jss.v096.i03>. Choice simulation and model estimation in power analyses are handled using the 'logitr' package by Helveston (2023) <doi:10.18637/jss.v105.i10>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

VignetteBuilder knitr

Depends R (>= 3.5.0)

Suggests here, knitr, testthat, tibble

Imports fastDummies, ggplot2, idefix (>= 1.1.0), logitr (>= 1.0.1), parallel, randtoolbox, rlang, tools, stats, utils

URL https://github.com/jhelvy/cbcTools, https://jhelvy.github.io/cbcTools/

BugReports https://github.com/jhelvy/cbcTools/issues

NeedsCompilation no

Author John Helveston [cre, aut, cph] (ORCID: <https://orcid.org/0000-0002-2657-9191>)

Repository CRAN

Date/Publication 2025-10-20 19:50:02 UTC

Contents

`cbc_choices` *Simulate choices for a survey design*

Description

Simulate choices for a survey design, either randomly or according to a utility model defined by user-provided prior parameters. When priors are provided, choices are simulated using the same probability computation framework as used in cbc_design() for consistency.

Usage

```
cbc_choices(design, priors = NULL)
```

Arguments

`design`	A `cbc_design` object created by `cbc_design()`
`priors`	A `cbc_priors` object created by `cbc_priors()`, or `NULL` (default) for random choices.

Value

Returns the input design with an additional `choice` column identifying the simulated choices.

Examples

```
library(cbcTools)

# Create profiles and design
profiles <- cbc_profiles(
  price = c(1, 2, 3),
  type = c("A", "B", "C"),
  quality = c("Low", "High")
)

design <- cbc_design(
  profiles = profiles,
  n_alts = 2,
  n_q = 4
)

# Simulate random choices (default)
choices_random <- cbc_choices(design)

# Create priors and simulate utility-based choices
priors <- cbc_priors(
  profiles = profiles,
  price = -0.1,
  type = c(0.5, 0.2),  # vs reference level
  quality = 0.3
)

choices_utility <- cbc_choices(design, priors = priors)
```

---

cbc_compare *Compare multiple choice experiment designs*

Description

This function compares multiple CBC designs across key quality metrics including D-error, balance, overlap, and structural characteristics. Useful for evaluating different design methods or parameter settings.

Usage

```
cbc_compare(..., metrics = "all", sort_by = "d_error", ascending = NULL)
```

Arguments

...	Any number of cbc_design objects to compare, separated by commas. Can be named for clearer output (e.g., random = design1, stochastic = design2).
metrics	Character vector specifying which metrics to compare. Options: "structure", "efficiency", "balance", "overlap", or "all" (default). Can specify multiple: c("efficiency", "balance")
sort_by	Character. Metric to sort designs by. Options: "d_error" (default), "balance", "overlap", "profiles_used", "generation_time", or "none"
ascending	Logical. If TRUE, sort in ascending order (lower is better). If FALSE, sort in descending order (higher is better). Default depends on metric.

Value

A cbc_comparison object containing comparison results, printed in a formatted table.

Examples

```
library(cbcTools)

# Create profiles
profiles <- cbc_profiles(
  price = c(1, 2, 3),
  type = c("A", "B", "C"),
  quality = c("Low", "High")
)

# Create different designs
design_random <- cbc_design(
  profiles = profiles,
  method = "random",
  n_alts = 2, n_q = 4
)

design_stochastic <- cbc_design(
  profiles = profiles,
  method = "stochastic",
  n_alts = 2, n_q = 4
)

# Compare designs
cbc_compare(design_random, design_stochastic)

# Named comparison with specific metrics
cbc_compare(
  Random = design_random,
  Stochastic = design_stochastic,
  metrics = c("efficiency", "balance"),
  sort_by = "d_error"
)
```

cbc_decode — *Convert dummy-coded CBC data back to categorical format*

Description

This function is depreciated. Use `cbc_encode()` instead

Usage

```
cbc_decode(data)
```

Arguments

`data`	A `cbc_design` or `cbc_choices` object with dummy-coded categorical variables

Value

The input object with categorical variables restored to their original format

cbc_design — *Generate survey designs for choice experiments (Updated Implementation)*

Description

This function creates experimental designs for choice-based conjoint experiments using multiple design approaches including optimization and frequency-based methods.

Usage

```
cbc_design(
  profiles,
  method = "random",
  priors = NULL,
  n_alts,
  n_q,
  n_resp = 100,
  n_blocks = 1,
  n_cores = NULL,
  no_choice = FALSE,
  label = NULL,
  balance_by = NULL,
  randomize_questions = TRUE,
  randomize_alts = TRUE,
  remove_dominant = FALSE,
  dominance_types = c("total", "partial"),
```

```
  dominance_threshold = 0.8,
  max_dominance_attempts = 50,
  max_iter = 50,
  n_start = 5,
  include_probs = FALSE,
  use_idefix = TRUE
)
```

Arguments

profiles	A data frame of class cbc_profiles created using cbc_profiles()
method	Choose the design method: "random", "shortcut", "minoverlap", "balanced", "stochastic", "modfed", or "cea". Defaults to "random"
priors	A cbc_priors object created by cbc_priors(), or NULL for random/shortcut designs
n_alts	Number of alternatives per choice question
n_q	Number of questions per respondent (or per block)
n_resp	Number of respondents (for random/shortcut designs) or 1 (for optimized designs that get repeated)
n_blocks	Number of blocks in the design. Defaults to 1
n_cores	Number of cores to use for parallel processing in the design search. Defaults to NULL, in which case it is set to the number of available cores minus 1.
no_choice	Include a "no choice" option? Defaults to FALSE
label	The name of the variable to use in a "labeled" design. Defaults to NULL
balance_by	Character vector of attribute names to balance sampling across. Ensures balanced representation across levels of specified attributes. Only compatible with "random", "shortcut", "minoverlap", and "balanced" methods. Cannot be used with labeled designs or D-optimal methods ("stochastic", "modfed", "cea"). Defaults to NULL
randomize_questions	Randomize question order for each respondent? Defaults to TRUE (optimized methods only)
randomize_alts	Randomize alternative order within questions? Defaults to TRUE (optimized methods only)
remove_dominant	Remove choice sets with dominant alternatives? Defaults to FALSE
dominance_types	Types of dominance to check: "total" and/or "partial"
dominance_threshold	Threshold for total dominance detection. Defaults to 0.8
max_dominance_attempts	Maximum attempts to replace dominant choice sets. Defaults to 50.
max_iter	Maximum iterations for optimized designs. Defaults to 50
n_start	Number of random starts for optimized designs. Defaults to 5

include_probs Include predicted probabilities in resulting design? Requires priors. Defaults to FALSE

use_idefix If TRUE (the default), the idefix package will be used to find optimal designs, which is faster. Only valid with "cea" and "modfed" methods.

Details

Design Methods:

The method argument determines the design approach used:

- "random": Creates designs by randomly sampling profiles for each respondent independently
- "shortcut": Frequency-based greedy algorithm that balances attribute level usage
- "minoverlap": Greedy algorithm that minimizes attribute overlap within choice sets
- "balanced": Greedy algorithm that maximizes overall attribute balance across the design
- "stochastic": Stochastic profile swapping with D-error optimization (first improvement found)
- "modfed": Modified Fedorov algorithm with exhaustive profile swapping for D-error optimization
- "cea": Coordinate Exchange Algorithm with attribute-by-attribute D-error optimization

Method Compatibility:

The table below summarizes method compatibility with design features:

Method	No choice?	Labeled designs?	Restricted profiles?	balance_by?	Blocking?	Interactions?	Dominan
"random"	Yes	Yes	Yes	Yes	No	Yes	Yes
"shortcut"	Yes	Yes	Yes	Yes	No	No	Yes
"minoverlap"	Yes	Yes	Yes	Yes	No	No	Yes
"balanced"	Yes	Yes	Yes	Yes	No	No	Yes
"stochastic"	Yes	Yes	Yes	No	Yes	Yes	Yes
"modfed"	Yes	Yes	Yes	No	Yes	Yes	Yes
"cea"	Yes	Yes	No	No	Yes	Yes	Yes

Design Quality Assurance:

All methods ensure the following criteria are met:

1. No duplicate profiles within any choice set
2. No duplicate choice sets within any respondent
3. If remove_dominant = TRUE, choice sets with dominant alternatives are eliminated (optimization methods only)

Balanced Sampling with balance_by:

The balance_by argument enables balanced sampling across specified attributes, solving the problem of attribute-specific features that create imbalanced designs. For example, consider an experiment on alternative vehicle powertrains with a "powertrain" attribute for gas and electric vehicles. If you had an "electric_vehicle_range" attribute, it should be 0 for non-electric powertrains, but using restrictions can lead to over-representation of electric vehicles. Using balance_by = "powertrain" ensures that each choice question samples proportionally from gas and electric powertrains, maintaining balance even when electric vehicles have additional attributes.

Multiple attributes can be balanced simultaneously using `balance_by = c("attr1", "attr2")`, which creates groups based on unique combinations of the specified attributes.

Method Details:

Random Method:
Creates designs where each respondent sees completely independent, randomly generated choice sets.

Greedy Methods (shortcut, minoverlap, balanced):
These methods use frequency-based algorithms that make locally optimal choices:

- **Shortcut**: Balances attribute level usage within questions and across the overall design
- **Minoverlap**: Minimizes attribute overlap within choice sets while allowing some overlap for balance
- **Balanced**: Maximizes overall attribute balance, prioritizing level distribution over overlap reduction

These methods provide good level balance without requiring priors or D-error calculations and offer fast execution suitable for large designs.

D-Error Optimization Methods (stochastic, modfed, cea):
These methods minimize D-error to create statistically efficient designs:

- **Stochastic**: Random profile sampling with first improvement acceptance
- **Modfed**: Exhaustive profile testing for best improvement (slower but thorough)
- **CEA**: Coordinate exchange testing attribute levels individually (requires full factorial profiles)

idefix Integration:

When `use_idefix = TRUE` (the default), the function leverages the highly optimized algorithms from the idefix package for 'cea' and 'modfed' design generation methods. This can provide significant speed improvements, especially for larger problems.

Key benefits of idefix integration:

- Faster optimization algorithms with C++ implementation
- Better handling of large candidate sets
- Optimized parallel processing
- Advanced blocking capabilities for multi-block designs

Value

A `cbc_design` object containing the experimental design

Examples

```
library(cbcTools)

# Create profiles for an apple choice experiment
profiles <- cbc_profiles(
    price = c(1, 1.5, 2, 2.5, 3),
    type = c("Fuji", "Gala", "Honeycrisp"),
    freshness = c("Poor", "Average", "Excellent")
)
```

```
# Basic random design
design_random <- cbc_design(
    profiles = profiles,
    n_alts = 3,
    n_q = 6,
    n_resp = 100
)

head(design_random)

# Inspect design
cbc_inspect(design_random)

# Greedy design with balanced frequency
design_balanced <- cbc_design(
    profiles = profiles,
    method = "balanced",
    n_alts = 3,
    n_q = 6,
    n_resp = 100
)

# Design with priors using D-optimal method
priors <- cbc_priors(
    profiles = profiles,
    price = -0.25,
    type = c("Gala" = 0.5, "Honeycrisp" = 1.0),
    freshness = c("Average" = 0.6, "Excellent" = 1.2)
)

design_optimal <- cbc_design(
    profiles = profiles,
    method = "stochastic",
    priors = priors,
    n_alts = 3,
    n_q = 6,
    n_resp = 100,
    n_start = 3
)

# Compare designs
cbc_compare(
    "Random" = design_random,
    "Balanced" = design_balanced,
    "D-optimal" = design_optimal
)
```

---

cbc_encode *Encode categorical variables in a CBC design*

Description

This function converts categorical variables between different coding schemes. Standard coding keeps categorical variables as-is (factor or character). Dummy coding uses a reference category (all zeros) with indicator variables for other levels. Effects coding uses -1 for the reference category to ensure coefficients sum to zero.

Usage

```
cbc_encode(data, coding = NULL, ref_levels = NULL)
```

Arguments

data	A cbc_design or cbc_choices object
coding	Character. Type of encoding: "standard", "dummy", or "effects". If NULL and ref_levels is NULL, data is returned unchanged. If NULL and ref_levels is specified, the current encoding is maintained.
ref_levels	Named list specifying reference levels for categorical variables. For example: list(powertrain = "Gasoline", brand = "A"). If NULL (default), uses the first level of each categorical variable as reference.

Value

The input object with specified encoding applied

Examples

```
library(cbcTools)

# Create profiles with categorical variables
profiles <- cbc_profiles(
  price = c(10, 20, 30),
  quality = c("Low", "Medium", "High"),
  brand = c("A", "B")
)

# Create design (defaults to standard coding)
design <- cbc_design(
  profiles = profiles,
  n_alts = 2,
  n_q = 4
)

# Convert to dummy coding
design_dummy <- cbc_encode(design, coding = "dummy")
head(design_dummy)

# Convert to effects coding
design_effects <- cbc_encode(design, coding = "effects")
head(design_effects)
```

```
# Convert back to standard
design_standard <- cbc_encode(design_dummy, coding = "standard")
head(design_standard)

# Custom reference levels with dummy coding
design_dummy2 <- cbc_encode(
  design,
  coding = "dummy",
  ref_levels = list(quality = "Medium", brand = "B")
)
head(design_dummy2)

# Update reference levels without changing encoding
design_updated <- cbc_encode(
  design_dummy,
  ref_levels = list(quality = "High")
)
head(design_updated)
```

`cbc_inspect` *Comprehensive design quality inspection*

Description

This function provides detailed inspection of choice experiment designs across multiple dimensions including design structure, efficiency metrics, attribute balance, overlap patterns, and variable encoding.

Usage

```
cbc_inspect(design, sections = "all", verbose = FALSE)
```

Arguments

`design`	A `cbc_design` or `cbc_choices` object created by `cbc_design()`
`sections`	Character vector specifying which sections to show. Options: "structure", "efficiency", "balance", "overlap", "encoding", or "all" (default). Can specify multiple: `c("balance", "overlap")`
`verbose`	Logical. If TRUE, shows additional technical details. If FALSE (default), shows simplified output.

Value

A `cbc_inspection` object containing the inspection results

Examples

```
library(cbcTools)

# Create profiles and design
profiles <- cbc_profiles(
  price = c(1, 2, 3),
  type = c("A", "B", "C"),
  quality = c("Low", "High")
)

design <- cbc_design(
  profiles = profiles,
  n_alts = 2,
  n_q = 4
)

# Inspect all sections (default) - prints automatically
cbc_inspect(design)

# Store results for later use
inspection <- cbc_inspect(design, sections = "balance")
inspection  # prints the same output

# Verbose output with technical details
cbc_inspect(design, verbose = TRUE)
```

---

cbc_power *Estimate power analysis for choice experiment designs*

Description

This function estimates the same model multiple times using different sample sizes to assess statistical power. It returns both the estimated models and a summary of coefficient estimates, standard errors, and power statistics.

Usage

```
cbc_power(
  data,
  outcome = "choice",
  obsID = "obsID",
  pars = NULL,
  randPars = NULL,
  n_breaks = 10,
  n_q = NULL,
  panelID = NULL,
  alpha = 0.05,
  return_models = FALSE,
```

```
  n_cores = NULL,
  ...
)
```

Arguments

data	A data frame containing choice data. Can be a cbc_choices object or any data frame with the required columns.
outcome	Name of the outcome variable column (1 for chosen, 0 for not). Defaults to "choice".
obsID	Name of the observation ID column. Defaults to "obsID".
pars	Names of the parameters to estimate. If NULL (default), will auto-detect from column names for cbc_choices objects.
randPars	Named vector of random parameters and their distributions ('n' for normal, 'ln' for log-normal). Defaults to NULL.
n_breaks	Number of sample size groups to test. Defaults to 10.
n_q	Number of questions per respondent. Auto-detected for cbc_choices objects if not specified.
panelID	Name of the panel ID column for panel data. Auto-detected as "respID" for multi-respondent cbc_choices objects.
alpha	Significance level for power calculations. Defaults to 0.05.
return_models	If TRUE, includes full model objects in returned list. Defaults to FALSE.
n_cores	Number of cores for parallel processing. Defaults to parallel::detectCores() - 1.
...	Additional arguments passed to logitr::logitr().

Value

A cbc_power object containing:

- power_summary: Data frame with sample sizes, coefficients, estimates, standard errors, t-statistics, and power
- models: List of estimated models (if return_models = TRUE)
- sample_sizes: Vector of sample sizes tested
- n_breaks: Number of breaks used
- alpha: Significance level used

Examples

```
library(cbcTools)

# Create profiles and design
profiles <- cbc_profiles(
  price = c(1, 2, 3),
  type = c("A", "B", "C"),
```

```
  quality = c("Low", "High")
)

design <- cbc_design(profiles, n_alts = 2, n_q = 6)

# Simulate choices
priors <- cbc_priors(profiles, price = -0.1, type = c(0.5, 0.2), quality = 0.3)
choices <- cbc_choices(design, priors)

# Run power analysis
power_results <- cbc_power(choices, n_breaks = 8)

# View results
print(power_results)
plot(power_results)
```

`cbc_priors` *Create prior specifications for CBC models*

Description

Creates a standardized prior specification object for use in CBC analysis functions like `cbc_choices()` and `cbc_design()`. Supports both fixed and random parameters, with flexible specification of categorical variable levels and interaction terms between fixed parameters.

Usage

```
cbc_priors(
  profiles,
  no_choice = NULL,
  n_draws = 100,
  draw_type = "halton",
  interactions = NULL,
  ...
)
```

Arguments

`profiles`	A data frame of profiles created by `cbc_profiles()`
`no_choice`	Prior specification for no-choice alternative. Can be: • A single numeric value for fixed no-choice utility • A `rand_spec()` object for random no-choice utility • `NULL` if no no-choice option (default)
`n_draws`	Number of draws for DB-error calculation if using Bayesian priors. Defaults to `100`
`draw_type`	Specify the draw type as a character: `"halton"` (the default) or `"sobol"` (recommended for models with more than 5 random parameters).

`interactions` A list of interaction specifications created by `int_spec()`. Only interactions between fixed (non-random) parameters are supported. Each interaction must specify the appropriate level(s) for categorical variables. Defaults to `NULL` (no interactions).

`...` Named arguments specifying priors for each attribute:

- For fixed parameters:
  - Continuous variables: provide a single numeric value
  - Categorical variables: provide either:
    - An unnamed vector of values one less than the number of levels (dummy coding)
    - A named vector mapping levels to coefficients (remaining level becomes reference)
- For random parameters: use `rand_spec()` to specify distribution, parameters, and correlations

Details

Fixed vs Random Parameters:

Fixed parameters assume all respondents have the same preference coefficients. Specify these as simple numeric values.

Random parameters assume preference coefficients vary across respondents according to a specified distribution. Use `rand_spec()` to define the distribution type, mean, and standard deviation.

Categorical Variable Specification:

For categorical variables, you can specify priors in two ways:

1. **Unnamed vector**: Provide coefficients for all levels except the first (which becomes the reference level). Order matters and should match the natural order of levels.
2. **Named vector**: Explicitly map coefficient values to specific levels. Any level not specified becomes the reference level.

Interaction Terms:

Use the `interactions` parameter with `int_spec()` to include interaction effects between attributes. Only interactions between fixed parameters are supported. For categorical variables involved in interactions, you must specify the relevant levels.

No-Choice Options:

When including a no-choice alternative, provide a `no_choice` parameter. This can be either a fixed numeric value or a `rand_spec()` for random no-choice utility.

Value

A structured prior specification object including parameter draws for random coefficients and interaction terms. This object contains:

- `pars`: Vector of mean parameter values
- `par_draws`: Matrix of parameter draws (if random parameters specified)
- `correlation`: Correlation matrix for random parameters (if applicable)

- `interactions`: List of interaction specifications
- `attrs`: Detailed attribute information
- Additional metadata for validation and compatibility checking

Examples

```
library(cbcTools)

# Create profiles for examples
profiles <- cbc_profiles(
  price = c(1, 1.5, 2, 2.5, 3),
  type = c('Fuji', 'Gala', 'Honeycrisp'),
  freshness = c('Poor', 'Average', 'Excellent')
)

# Example 1: Simple fixed priors
priors_fixed <- cbc_priors(
  profiles = profiles,
  price = -0.25, # Negative = prefer lower prices
  type = c(0.5, 1.0), # "Fuji" is reference level
  freshness = c(0.6, 1.2) # "Poor" reference level
)

# Example 2: Named categorical priors (more explicit)
priors_named <- cbc_priors(
  profiles = profiles,
  price = -0.25,
  type = c("Gala" = 0.5, "Honeycrisp" = 1.0),  # "Fuji" is reference
  freshness = c("Average" = 0.6, "Excellent" = 1.2)  # "Poor" is reference
)

# Example 3: Random parameters - normal distributions for "price" and "freshness"
priors_random <- cbc_priors(
  profiles = profiles,
  price = rand_spec(
    dist = "n",
    mean = -0.25,
    sd = 0.1
  ),
  type = c(0.5, 1.0),
  freshness = rand_spec(
    dist = "n",
    mean = c(0.6, 1.2),
    sd = c(0.1, 0.1)
  )
)

# Example 4: Correlated random parameters
priors_correlated <- cbc_priors(
  profiles = profiles,
  price = rand_spec(
    dist = "n",
```

```
    mean = -0.1,
    sd = 0.05,
    correlations = list(
      cor_spec(
        with = "type",
        with_level = "Honeycrisp",
        value = 0.3
      )
    )
  ),
  type = rand_spec(
    dist = "n",
    mean = c("Gala" = 0.1, "Honeycrisp" = 0.2),
    sd = c("Gala" = 0.05, "Honeycrisp" = 0.1)
  ),
  freshness = c(0.1, 0.2)
)

# Example 5: With interaction terms
priors_interactions <- cbc_priors(
  profiles = profiles,
  price = -0.25,
  type = c("Fuji" = 0.5, "Honeycrisp" = 1.0),
  freshness = c("Average" = 0.6, "Excellent" = 1.2),
  interactions = list(
    # Price sensitivity varies by apple type
    int_spec(
      between = c("price", "type"),
      with_level = "Fuji",
      value = 0.1
    ),
    int_spec(
      between = c("price", "type"),
      with_level = "Honeycrisp",
      value = 0.2
    ),
    # Type preferences vary by freshness
    int_spec(
      between = c("type", "freshness"),
      level = "Honeycrisp",
      with_level = "Excellent",
      value = 0.3
    )
  )
)

# Example 6: Including no-choice option
priors_nochoice_fixed <- cbc_priors(
  profiles = profiles,
  price = -0.25,
  type = c(0.5, 1.0),
  freshness = c(0.6, 1.2),
  no_choice = -0.5 # Negative values make no-choice less attractive
```

```
)

# Example 7: Random no-choice
priors_nochoice_random <- cbc_priors(
  profiles = profiles,
  price = -0.25,
  type = c(0.5, 1.0),
  freshness = c(0.6, 1.2),
  no_choice = rand_spec(dist = "n", mean = -0.5, sd = 0.2)
)

# View the priors
priors_fixed
priors_random
```

`cbc_profiles` *Make a data frame of all combinations of attribute levels*

Description

This function creates a data frame of of all possible combinations of attribute levels.

Usage

```
cbc_profiles(...)
```

Arguments

`...`	Any number of named vectors defining each attribute and their levels, e.g. `price = c(1, 2, 3)`. Separate each vector by a comma.

Value

A data frame of all possible combinations of attribute levels with class `cbc_profiles`.

Examples

```
library(cbcTools)

# Generate all profiles for a simple conjoint experiment about apples
profiles <- cbc_profiles(
  price     = c(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5),
  type      = c("Fuji", "Gala", "Honeycrisp"),
  freshness = c('Poor', 'Average', 'Excellent')
)
```

cbc_restrict *Obtain a restricted set of profiles*

Description

This function returns a restricted set of profiles as a data frame.

Usage

```
cbc_restrict(profiles, ...)
```

Arguments

profiles	A data frame of class cbc_profiles created using the cbc_profiles() function.
...	Any number of restricted pairs of attribute levels, defined as pairs of logical expressions separated by commas. For example, the restriction type == 'Fuji' & freshness == 'Poor' will eliminate profiles such that "Fuji" type apples will never be shown with "Poor" freshness.

Value

A restricted set of profiles as a data frame with class cbc_profiles.

Examples

```
library(cbcTools)

# Generate all profiles for a simple conjoint experiment about apples
profiles <- cbc_profiles(
  price     = c(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5),
  type      = c("Fuji", "Gala", "Honeycrisp"),
  freshness = c('Poor', 'Average', 'Excellent')
)

# Obtain a restricted subset of profiles based on pairs of logical
# expressions. The example below contains the following restrictions:

# - `"Gala"` apples will not be shown with the prices `1.5`, `2.5`, & `3.5`.
# - `"Honeycrisp"` apples will not be shown with prices less than `2`.
# - `"Honeycrisp"` apples will not be shown with the `"Poor"` freshness.
# - `"Fuji"` apples will not be shown with the `"Excellent"` freshness.

profiles_restricted <- cbc_restrict(
    profiles,
    type == "Gala" & price %in% c(1.5, 2.5, 3.5),
    type == "Honeycrisp" & price > 2,
    type == "Honeycrisp" & freshness == "Poor",
    type == "Fuji" & freshness == "Excellent"
)
```

cor_spec *Create a correlation specification for random parameters*

Description

Create a correlation specification for random parameters

Usage

```
cor_spec(with, value, level = NULL, with_level = NULL)
```

Arguments

with	Character. Name of attribute to correlate with
value	Numeric. Correlation value between -1 and 1
level	Character. For categorical variables, specific level to correlate from
with_level	Character. For categorical variables, specific level to correlate with

Value

A correlation specification list

int_spec *Create an interaction specification for fixed parameters*

Description

Create an interaction specification for fixed parameters

Usage

```
int_spec(between, value, level = NULL, with_level = NULL)
```

Arguments

between	Character vector of length 2 specifying the two attributes to interact
value	Numeric. Interaction coefficient value
level	Character. For categorical variables, specific level of first attribute
with_level	Character. For categorical variables, specific level of second attribute

Value

An interaction specification list

Examples

```
# Continuous * continuous interaction
int_spec(between = c("price", "weight"), value = 0.1)

# Continuous * categorical interactions (must specify categorical level)
int_spec(between = c("price", "type"), with_level = "Fuji", value = 0.15)
int_spec(between = c("price", "type"), with_level = "Gala", value = 0.05)

# Categorical * categorical interactions (must specify both levels)
int_spec(between = c("type", "freshness"),
         level = "Fuji", with_level = "Poor", value = -0.2)
```

---

plot.cbc_power *Plot method for cbc_power objects*

Description

Plot method for cbc_power objects

Usage

```
## S3 method for class 'cbc_power'
plot(x, type = "power", power_threshold = 0.8, ...)
```

Arguments

x	A cbc_power object
type	Type of plot: "power" for power curves or "se" for standard error curves
power_threshold	Power threshold for horizontal reference line (only for power plots). Defaults to 0.8
...	Additional arguments passed to ggplot

Value

Returns a ggplot2 object (class: "gg", "ggplot") that can be further customized, saved, or displayed. The plot visualizes either statistical power curves or standard error curves across different sample sizes for each parameter in the power analysis, with appropriate axis labels, legends, and reference lines.

plot_compare_power *Compare power across multiple designs*

Description

Compare power across multiple designs

Usage

```
plot_compare_power(..., type = "power", power_threshold = 0.8)
```

Arguments

...	Named cbc_power objects to compare
type	Type of plot: "power" for power curves or "se" for standard error curves
power_threshold	Power threshold for horizontal reference line (only for power plots). Defaults to 0.8

Value

A ggplot object comparing power curves

print.cbc_choices *Print method for cbc_choices objects*

Description

Print method for cbc_choices objects

Usage

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

Arguments

x	A cbc_choices object
...	Additional arguments passed to print

Value

Returns the input `cbc_choices` object invisibly (class: c("cbc_choices", "data.frame")). This function is called for its side effect of printing a formatted summary of the CBC choice data to the console, including choice task structure, simulation details, choice rates by alternative, and a preview of the choice data.

print.cbc_comparison *Print method for cbc_comparison objects*

Description

Print method for cbc_comparison objects

Usage

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

Arguments

x	A cbc_comparison object
...	Additional arguments passed to print

Value

Returns the input cbc_comparison object invisibly (class: c("cbc_comparison", "list")). This function is called for its side effect of printing a formatted comparison table of multiple CBC designs to the console, including design metrics, performance rankings, and interpretation guidelines.

print.cbc_design *Concise print method for cbc_design objects*

Description

Concise print method for cbc_design objects

Usage

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

Arguments

x	A cbc_design object
...	Additional arguments passed to print

Value

Returns the input cbc_design object invisibly (class: c("cbc_design", "data.frame")). This function is called for its side effect of printing a concise summary of the CBC design to the console, including design method, structure, D-error metrics, profile usage, and a preview of the design data.

print.cbc_inspection *Print method for cbc_inspection objects*

Description

Print method for cbc_inspection objects

Usage

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

Arguments

x	A cbc_inspection object
...	Additional arguments passed to print

Value

Returns the input cbc_inspection object invisibly (class: c("cbc_inspection", "list")). This function is called for its side effect of printing a comprehensive inspection report of the CBC design to the console, including sections on design structure, efficiency metrics, attribute balance, overlap analysis, and variable encoding.

print.cbc_power *Print method for cbc_power objects*

Description

Print method for cbc_power objects

Usage

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

Arguments

x	A cbc_power object
...	Additional arguments passed to print

Value

Returns the input cbc_power object invisibly (class: c("cbc_power", "list")). This function is called for its side effect of printing a formatted summary of the CBC power analysis results to the console, including sample size ranges, significance levels, parameter summaries, and power estimates across different sample sizes.

print.cbc_priors *Print method for cbc_priors objects*

Description

Print method for cbc_priors objects

Usage

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

Arguments

x	A cbc_priors object
...	Additional arguments passed to print

Value

Returns the input cbc_priors object invisibly (class: c("cbc_priors", "list")). This function is called for its side effect of printing a formatted summary of the CBC priors specifications to the console, including parameter types, distributions, means, standard deviations, and any correlation structures.

print.cbc_profiles *Print method for cbc_profiles objects*

Description

Print method for cbc_profiles objects

Usage

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

Arguments

x	A cbc_profiles object
...	Additional arguments passed to print

Value

Returns the input cbc_profiles object invisibly (class: c("cbc_profiles", "data.frame")). This function is called for its side effect of printing a formatted summary of the CBC profiles object to the console, including attribute information, profile counts, any applied restrictions, and a preview of the data.

rand_spec *Create a random parameter specification*

Description

Create a random parameter specification

Usage

```
rand_spec(dist = "n", mean, sd, correlations = NULL)
```

Arguments

dist	Character. Distribution type: "n" for normal, "ln" for log-normal, or "cn" for censored normal
mean	Numeric. Mean parameter value(s)
sd	Numeric. Standard deviation parameter value(s)
correlations	List of correlation specifications created by cor_spec()

Value

A random parameter specification list

summary.cbc_power *Summary method for cbc_power objects*

Description

Summary method for cbc_power objects

Usage

```
## S3 method for class 'cbc_power'
summary(object, power_threshold = 0.8, ...)
```

Arguments

object	A cbc_power object
power_threshold	Minimum power threshold to report sample size requirements
...	Additional arguments

Value

Returns the input `cbc_power` object invisibly (class: c("cbc_power", "list")). This function is called for its side effect of printing a detailed summary to the console showing sample size requirements for achieving specified power thresholds for each parameter, including exact power levels and standard errors at the required sample sizes.

Index