

Package: causalgenerics (via r-universe)

October 1, 2026

Title Shared Generics for the 'r-causal' Ecosystem

Version 0.1.0.9000

Description A home for the S3 generics shared across the 'r-causal' ecosystem, including 'propensity', 'halfmoon', 'positively', and 'balancing'. Owning the generic definitions in one place lets those packages register methods without masking one another when several are attached at once. The generics cover inverse probability weighted estimation, effective sample size, and the metadata carried by causal weight vectors, and the package supplies the abstract causal weight class and the result class those estimates are returned in, together with the methods both classes carry, so that the ecosystem packages inherit them rather than writing their own. The design follows that of the 'generics' package, which provides commonly used S3 generics for the same purpose: letting packages share a single definition instead of each defining its own.

License MIT + file LICENSE

URL <https://github.com/r-causal/causalgenerics>,
<https://r-causal.github.io/causalgenerics/>

BugReports <https://github.com/r-causal/causalgenerics/issues>

Depends R (>= 4.1.0)

Imports cli, rlang, stats, vctrs

Suggests testthat (>= 3.2.0), withr

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

Repository <https://r-causal.r-universe.dev>

Date/Publication 2026-09-01 13:47:00 UTC

RemoteUrl <https://github.com/r-causal/causalgenerics>
RemoteRef HEAD
RemoteSha 7a98f37cc623732c81075502600412dc459f481d

Contents

causal-weights	2
causal_wts_ptype2	4
check_forced_type	5
detect_exposure_type	7
ess	8
exposure-type-helpers	10
ipw	11
ipw-accessors	13
ipw-modes	18
ipw-pooled-methods	21
joint-exposure-accessors	27
joint_exposure	28
match_exposure_type	30
new_causal_wts	32
new_ipw	33
new_ipw_model	40
pool_ipw	41
Index	47

causal-weights	<i>Causal weight accessors</i>
----------------	--------------------------------

Description

Generics for inspecting and setting the metadata carried by causal weight vectors.

- `is_causal_wt()` tests whether an object is a causal weight vector.
- `estimand()` returns the causal estimand the weights target, such as "ate" or "att".
- `estimand<-( )` sets the causal estimand.

Usage

```
is_causal_wt(x, ...)  
  
estimand(x, ...)  
  
estimand(x, ...) <- value
```

Arguments

<code>x</code>	An object to inspect or modify. These generics dispatch on this argument.
<code>...</code>	Arguments passed to methods.
<code>value</code>	The causal estimand to assign.

Details

This package defines the generics and the abstract weight class they are written against. It supplies a method for each generic on `causal_wts`, so a concrete class built with `new_causal_wts()` inherits all three. The concrete classes themselves live in the packages that own them, such as `propensity`, as do methods for any object that is not a `causal_wts` vector. `is_causal_wt()` defaults to `FALSE` so that any object that is not a recognized causal weight vector reports as much; `estimand()` and `estimand<-( )` have no meaningful default and signal an error when no method is registered for the object.

Value

`is_causal_wt()` returns a single logical value. `estimand()` returns the estimand, typically a character string, or `NULL` when none is recorded. `estimand<-( )` returns `x` with the estimand updated.

See Also

`new_causal_wts()` for the class these generics have methods for, and the `propensity` package for concrete weight classes. `ipw()` estimates an effect from a set of weights, and the result it returns answers `estimand()` with the estimand those weights targeted, so the same accessor reads the weights and the estimate made from them.

Examples

```

wts <- new_causal_wts(
  c(1.2, 0.8, 1.5),
  subclass = "cg_toy_wts",
  estimand = "ate"
)

is_causal_wt(wts)
estimand(wts)

# The estimand is metadata on the vector, so it can be replaced in place.
estimand(wts) <- "att"
estimand(wts)

# An object that is not a causal weight vector reports `FALSE` rather than
# erroring.
is_causal_wt(1:3)

# `estimand()` has no meaningful default, so it errors instead.
try(estimand(1:3))

```

causal_wts_ptype2	<i>Common type of two causal weight vectors of the same class</i>
-------------------	---

Description

`causal_wts_ptype2()` supplies the one coercion rule that concrete weight classes share: two causal weight vectors combine only when their estimands are identical, and otherwise the common type is a plain double. Concrete classes call it from inside their own `vec_trs::vec_ptype2()` method for the pairing of two vectors of that class.

Usage

```
causal_wts_ptype2(x, y, ptype, on_downgrade)
```

Arguments

<code>x, y</code>	The two weight vectors being combined, as passed to <code>vec_ptype2()</code> .
<code>ptype</code>	The common type to return when the estimands agree. Returned exactly as supplied and never inspected, and evaluated only on that path.
<code>on_downgrade</code>	A function of no arguments, called once when the estimands differ. Its value is discarded; it exists so the caller can signal a condition of its own choosing.

Details

This is a helper rather than a method, and it cannot become one. `vec_trs` resolves `vec_ptype2()` with an exact lookup on `class(x)[[1]]` and never walks the rest of the class vector, so a method registered on `causal_wts` is never found for a concrete subclass. Each package therefore keeps its own `vec_ptype2.<cls>.<cls>` method and calls this from inside it:

```
vec_ptype2.bw.bw <- function(x, y, ...) {
  causal_wts_ptype2(
    x,
    y,
    new_bw(estimand = estimand(x)),
    on_downgrade = function() warn_bw_downgrade("bw")
  )
}
```

The estimands are read through `estimand()`, so a class that computes its estimand rather than storing it is compared on the value its method returns. Two vectors that record no estimand are compatible, which is the ordinary case for continuous-exposure weights rather than an edge case.

`ptype` is evaluated only on the compatible path. A caller's prototype is usually a live constructor call over metadata the two vectors do not agree on, so the downgrade path never forces it.

The condition belongs to the caller. This function signals nothing of its own: `on_downgrade()` runs exactly once when the estimands differ, and whatever it signals reaches the user unchanged. The

concrete classes downstream disagree on the class of that condition and on whether the downgrade warns at all, so choosing one here would break them.

The scope is one pairing and one rule, which is as much as the concrete classes share. A class that checks nothing beyond the estimand hands over its whole method, as balancing does. A class that checks further metadata can use this for the estimand alone and keep its own control flow: propensity compares five more fields after the estimand, each with its own message and its own bail-out. Such a caller can pass a sentinel as ptype and carry on with its remaining checks when that sentinel comes back.

Value

ptype when estimand(x) and estimand(y) are identical, and double() otherwise.

See Also

`new_causal_wts()` for the class this rule is written against, and `estimand()` for the accessor it compares.

Examples

```
ate <- new_causal_wts(c(1.2, 0.8), subclass = "my_wts", estimand = "ate")
att <- new_causal_wts(c(1.0, 1.5), subclass = "my_wts", estimand = "att")

# Matching estimands give back the prototype the caller built.
causal_wts_ptype2(
  ate,
  ate,
  new_causal_wts(subclass = "my_wts", estimand = estimand(ate)),
  on_downgrade = function() message("the estimands differ")
)

# Differing estimands run the callback and drop to a bare double.
causal_wts_ptype2(
  ate,
  att,
  new_causal_wts(subclass = "my_wts", estimand = estimand(ate)),
  on_downgrade = function() message("the estimands differ")
)
```

check_forced_type

Refuse an exposure type the data cannot carry

Description

`check_forced_type()` is the structural check on a declared exposure type. A declaration wins over the detection heuristics wherever the data can carry it, so this refuses only the declarations that no data of this shape could support.

Usage

```
check_forced_type(
  forced,
  .exposure,
  arg = ".exposure",
  call = rlang::caller_env()
)
```

Arguments

<code>forced</code>	The type the caller declared: "binary", "categorical", or "continuous". "auto" is a request for detection rather than a declaration, and is refused here along with anything else that is not one of the three.
<code>.exposure</code>	The exposure vector to classify.
<code>arg</code>	The name the refusal gives the exposure argument.
<code>call</code>	The environment the refusal reports as its calling context.

Details

Two declarations can be contradicted. A binary exposure takes exactly two observed values, and a continuous one is numeric. Every vector can be read as the set of levels it takes, so a categorical declaration is always possible and always passes.

Everything else stands. A numeric dose taking fifteen values declared continuous is fitted as a dose even though detection would read it as categorical, and a numeric exposure taking two values declared continuous is the caller's decision. A check that refused those would put the unique-value heuristic in charge of a reading the caller had already made.

The refusal quotes what detection would have read, since dropping the declaration is the route out of it. That reading is computed and not announced: the caller declared a type, so an alert naming a different one would read as the answer rather than as the alternative.

The refusal names `exposure_type` as the argument the declaration came from, which is what the ecosystem calls it.

Value

NULL, invisibly. The function is called for the refusal.

See Also

[match_exposure_type\(\)](#), which resolves the declaration this checks, and [detect_exposure_type\(\)](#), which supplies the reading it quotes.

Examples

```
# A declaration the data can carry passes silently.
check_forced_type("binary", c(0, 1, NA, 1))

# So does one detection would have read the other way.
```

```

check_forced_type("continuous", rep_len(1:15, 300))

# A binary exposure takes two values, and this one takes three.
try(check_forced_type("binary", factor(c("low", "medium", "high"))))

# A continuous exposure is numeric.
try(check_forced_type("continuous", c("a", "b", "c")))

```

detect_exposure_type *Detect the type of an exposure*

Description

detect_exposure_type() reads an exposure vector and reports whether the analysis it feeds should treat it as binary, categorical, or continuous. It is the reading the r-causal packages take when a caller declares no type, so the classification lives in one place and every package that needs it gives the same answer for the same column.

Usage

```

detect_exposure_type(
  .exposure,
  arg = ".exposure",
  announce = TRUE,
  call = rlang::caller_env()
)

```

Arguments

.exposure	The exposure vector to classify.
arg	The name the announcement gives the exposure argument. A function whose exposure argument is named something else passes its own name, so the announcement never points at an argument the caller does not have.
announce	Should the detected type be announced? The announcement reports which branch the analysis took, so a caller passes FALSE where that is not worth reporting, such as one detecting several exposures at once. It is suppressed for every call, whatever this argument says, when options(causalgenerics.quiet = TRUE).
call	The environment a refusal reports as its calling context. Detection raises none of its own; the argument is here so that a caller threading its own call through the resolver passes it the same way to every part of it.

Details

The reading is taken in this order. A vector with exactly two observed values is binary, whatever its type. A factor or character vector taking more than two observed values is categorical, and one taking fewer is binary, so a single-level factor is read as a degenerate binary exposure rather than as a categorical one. Any other vector is categorical when `is_categorical()` finds few enough distinct values in it, and continuous otherwise.

A missing value is not a level. Every count is taken over the observed values, so an exposure recorded with missing values is read as the exposure it is: two observed values are binary however many observations are missing, and a categorical exposure reports the categories it has rather than one more.

Detection classifies what it is handed and refuses nothing. An exposure with one observed value, and one with none, each leave the branches above with a type rather than an error, because what a degenerate exposure means depends on what the calling function is about to fit with it. That refusal belongs to the calling function. `check_forced_type()` is the structural check for a type the caller declared.

Value

A single string: "binary", "categorical", or "continuous".

See Also

`match_exposure_type()`, which resolves a declared type against the types a function supports and calls this when the caller asked for detection.

Examples

```
# Two observed values are binary, and the missing one is not a third value.
detect_exposure_type(c(0, 1, NA, 1))

# More than two levels are categorical.
detect_exposure_type(factor(c("low", "medium", "high")))

# A numeric exposure taking many distinct values is continuous.
detect_exposure_type(seq_len(300) / 300)

# A function whose exposure argument is named otherwise says so.
detect_exposure_type(c(0, 1), arg = ".exposures")

# `announce = FALSE` returns the reading without reporting it.
detect_exposure_type(c(0, 1), announce = FALSE)
```

Description

`ess()` is the generic for the effective sample size, a summary of how much information a set of weights retains relative to an unweighted sample. When weights vary substantially, the effective sample size can be much smaller than the number of observations.

Usage

```
ess(x, ...)

## Default S3 method:
ess(x, na.rm = FALSE, ...)
```

Arguments

<code>x</code>	An object to compute the effective sample size for, such as a weight vector or a fitted model. <code>ess()</code> dispatches on this argument.
<code>...</code>	Arguments passed to methods.
<code>na.rm</code>	Should missing weights be dropped before computing? Missing weights otherwise make the result NA.

Details

The default method computes the Kish effective sample size, $(\sum w)^2 / \sum w^2$, for any numeric vector. Every weight class in this ecosystem is a double underneath, and `is.numeric()` is TRUE for one, so the default is the whole implementation for weights and a concrete weight class needs no method of its own. A method would be worth registering only for an object that needs something other than the Kish formula, such as a fitted model, where it might summarize by group and return a tibble with one row per group.

The default errors when `x` is not numeric, which includes NULL, rather than returning a value computed from nothing. Numeric input whose quotient is $0 / 0$ still returns NaN, which covers a zero-length vector and weights that are all zero: both make the sum and the sum of squares zero, and neither has an effective sample size to report.

The generic stays minimal, taking the object and `...`, so that a method declares whatever further arguments it needs. A fitted-model method, for example, takes the variable to group by that way.

Value

The default method returns a single number. Other methods return whatever suits the object they are written for; a fitted-model method may return a tibble.

See Also

`halfmoon:::check_ess()`, which reports the effective sample size of one or more weighting columns in a data frame, optionally by exposure group. It is an ordinary function rather than an `ess()` method, so it does not go through this generic.

Examples

```
# Equal weights carry as much information as an unweighted sample.
ess(rep(0.5, 10))

# Weights that vary carry much less.
ess(c(rep(0.5, 9), 10))
```

exposure-type-helpers *Read the values an exposure takes*

Description

The three readings `detect_exposure_type()` classifies from, exported so that a package taking its own decision from the same facts reads them the same way.

`observed_values()` returns the distinct values a vector takes. `has_two_levels()` reports whether there are exactly two of them. `is_categorical()` applies the unique-value heuristic that separates a categorical numeric exposure from a continuous one.

Usage

```
observed_values(.x)

has_two_levels(.x)

is_categorical(.exposure)
```

Arguments

<code>.x</code>	A vector.
<code>.exposure</code>	An exposure vector.

Details

A missing value is not a level. `observed_values()` drops the missing values before taking the distinct ones, and that is what the other two count, so an exposure recorded with missing values is read as the exposure it is. Counting the missing values as a value of their own would give a binary exposure three levels, which takes it off the binary path entirely, and would make a categorical one look as though it had one category more than it has.

`is_categorical()` treats a vector as categorical when the number of values it takes is less than 20 percent of the number of observations that are not missing. The threshold is strict, so a ratio of exactly 0.2 is not categorical. An exposure with nothing observed has no ratio to take and is not categorical, which leaves it on the continuous branch of detection rather than returning a missing value into the classification.

The distinct values are returned in the order they first appear rather than sorted, since they are only ever counted and sorting them would impose an ordering on a categorical exposure that nothing else here honors. The values keep the type they came in as, so the observed values of a factor are a factor over the same declared levels.

Value

`observed_values()` returns the distinct observed values, as a vector of the same type as its input. `has_two_levels()` and `is_categorical()` each return a single logical value.

See Also

[detect_exposure_type\(\)](#), which combines these into a type.

Examples

```
observed_values(c(0, 1, NA, 1))
has_two_levels(c(0, 1, NA, 1))

# 15 values over 300 observations is categorical; 300 over 300 is not.
is_categorical(rep_len(1:15, 300))
is_categorical(seq_len(300) / 300)
```

ipw

*Inverse probability weighted estimation***Description**

`ipw()` is the generic for bring-your-own-model inverse probability weighted estimation of causal effects. A method takes a fitted weighting or propensity score model together with a fitted weighted outcome model and returns causal effect estimates with standard errors that account for the two-step estimation process.

Usage

```
ipw(wt_mod, outcome_mod, ...)
```

Arguments

<code>wt_mod</code>	The weighting object that produced the weights, for example a fitted propensity score model. <code>ipw()</code> dispatches on this argument.
<code>outcome_mod</code>	A fitted weighted outcome model.
<code>...</code>	Arguments passed to methods.

Details

This package defines the generic and the shared result class its methods return. The methods themselves live in the packages that own the relevant model classes, such as propensity for propensity score models and balancing for balancing weight fits. Dispatch is on `wt_mod`, the weighting object; each method documents the outcome model classes and further arguments it accepts.

A method builds its return value with [new_ipw\(\)](#) rather than a result object of its own. The field names and their order are a cross-package contract, so that an IPW estimate reads the same way whichever package produced it, and constructing through [new_ipw\(\)](#) is also what gives a method the shared `print()` and `as.data.frame()` methods.

Value

An object of class `ipw`, holding the causal effect estimates and their standard errors alongside the models they came from. See `new_ipw()` for the components and their order.

See Also

`new_ipw()`, the constructor every method returns through, and the propensity and balancing packages for methods.

Examples

```
# A confounded toy data set: the confounder `x` raises both the chance of
# exposure `z` and the risk of outcome `y`. Within either level of `x` the
# risk difference is 0.25.
dat <- data.frame(
  x = rep(c(0, 1), each = 10),
  z = c(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0),
  y = c(1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0)
)

# The crude comparison ignores `x` and overstates that risk difference.
coef(lm(y ~ z, data = dat))["z"]

# A weighting object of a toy class, standing in for the fitted weighting
# models that the methods in other packages dispatch on. Weighting by the
# inverse probability of the exposure received breaks the dependence of `z`
# on `x`, so the weighted outcome model recovers the 0.25 risk difference.
ps <- fitted(glm(z ~ x, family = binomial(), data = dat))
wt_mod <- structure(
  list(wts = ifelse(dat$x == 1, 1 / ps, 1 / (1 - ps))),
  class = "cg_toy_model"
)

# A method for that class. It reads the exposure coefficient of the weighted
# outcome model as a risk difference and returns through `new_ipw()`, which is
# what every `ipw()` method does. Inference is normal-based throughout, as the
# `z` column of the estimates contract calls for. The standard error is the
# model-based one the weighted fit reports; a real method uses a variance
# estimator that also accounts for the weights having been estimated.
ipw.cg_toy_model <- function(wt_mod, outcome_mod, ...) {
  coefs <- summary(outcome_mod)$coefficients["z", ]
  estimate <- coefs[["Estimate"]]
  std_err <- coefs[["Std. Error"]]
  z <- estimate / std_err
  half_width <- qnorm(0.975) * std_err
  new_ipw(
    estimand = "ate",
    wt_mod = wt_mod,
    outcome_mod = outcome_mod,
    estimates = data.frame(
      effect = "rd",
      estimate = estimate,
```

```

      std.err = std_err,
      z = z,
      ci.lower = estimate - half_width,
      ci.upper = estimate + half_width,
      conf.level = 0.95,
      p.value = 2 * pnorm(-abs(z))
    ),
    se_method = "model-based",
    fit = NULL
  )
}

outcome_mod <- lm(y ~ z, data = dat, weights = wt_mod$wts)

# Dispatch on the weighting object finds the method.
ipw(wt_mod, outcome_mod)

```

ipw-accessors

Model accessors for an inverse probability weighted result

Description

The accessors a fitted model answers to, defined for the class `new_ipw()` constructs.

- `coef()` returns the effect estimates.
- `vcov()` returns the covariance of those estimates.
- `confint()` returns their confidence limits.
- `nobs()` returns the number of observations the outcome model was fitted on.
- `df.residual()` returns the residual degrees of freedom the fitted variance object reports.
- `weights()` returns the weights the outcome model was fitted with.
- `model.frame()` returns the outcome model's model frame.
- `estimand()` returns the estimand the weights targeted.

Usage

```

## S3 method for class 'ipw'
coef(object, ..., effects = NULL)

## S3 method for class 'ipw'
vcov(object, ..., effects = NULL)

## S3 method for class 'ipw'
confint(object, parm, level = 0.95, ..., effects = NULL)

## S3 method for class 'ipw'
nobs(object, ...)

```

```
## S3 method for class 'ipw'
df.residual(object, ...)

## S3 method for class 'ipw'
weights(object, ...)

## S3 method for class 'ipw'
model.frame(formula, ...)

## S3 method for class 'ipw'
estimand(x, ...)
```

Arguments

<code>object</code>	An ipw object.
<code>...</code>	Further arguments. These methods ignore them.
<code>effects</code>	The reading to report, either "marginal" or "conditional". NULL, the default, reports the reading the result records; any other value overrides it for the one call and leaves the result as it is. A reading the result does not support is refused with an error of class <code>causalgenerics_unsupported_reading</code> , which is where as_marginal() and as_conditional() refuse it too.
<code>parm</code>	The rows to report an interval for, given either as the labels of the reported surface or as their positions in it. Missing means all of them.
<code>level</code>	The confidence level. At the level every row of the result stores, the stored limits are returned; at any other level, and for a frame whose rows do not agree on one, every row is recomputed.
<code>formula</code>	An ipw object. The <code>model.frame()</code> generic in stats names its first argument <code>formula</code> , and the method matches it.
<code>x</code>	An ipw object. The <code>estimand()</code> generic names its first argument <code>x</code> , and the method matches it.

Details

These methods live here for the reason `print()` does. Two packages each registering `coef.ipw()` would collide in the shared S3 method table, and a caller writing against a result would then get whichever package was installed last rather than the contract.

Everything they read is part of the [new_ipw\(\)](#) contract: the `estimates` frame, the `ipw_vcov` attribute attached to it, `estimand`, `outcome_mod`, `fit`, and the `effects` field the section below describes. They never branch on `se_method`, since the fields they read already hold the result of whatever computation that names, and they reach into `fit` only through ordinary S3 dispatch. A package whose variance object is a bare list rather than a fitted model therefore uses them unchanged.

Rows are named by the effect labels the [new_ipw\(\)](#) contract defines, so the name `coef()` gives an estimate is the one `vcov()` and `confint()` use for it and the one `print()` labels its row with.

Value

`coef()` returns a named numeric vector of effect estimates, one element per row of the estimates frame, named by effect label. In the conditional reading it returns the outcome model's coefficients as that model reports them, names included.

`vcov()` returns the square numeric covariance matrix of those estimates, with the effect labels as dimnames on both margins, exactly as the fitting package attached it. A result that records no such matrix raises an error of class `causalgenerics_no_vcov` rather than returning one built from the standard errors, which would report the effects as uncorrelated. In the conditional reading it returns the corrected covariance of the outcome model's coefficients, in the order `coef()` reports them whatever order it was attached in. An outcome model that was never wrapped raises an error of class `causalgenerics_no_conditional_vcov`, a wrapped model that no longer carries the covariance raises an error of class `causalgenerics_no_vcov_ipw_model`, and a covariance whose labels cannot be paired with the coefficients raises an error of class `causalgenerics_conditional_vcov_mismatch`.

`confint()` returns a matrix with one row per effect parm selects, in the order parm gives them, and two columns holding the lower and upper limit. The rows are named by effect label and the columns by the two tail probabilities as percentages, the way the `confint()` methods in **stats** name theirs. A character parm that names an effect the result does not report raises an error of class `causalgenerics_invalid_argument`.

The limits the estimates frame stores are returned rather than rebuilt when every row of the frame records the level asked for, and every row is rebuilt from the estimate and its standard error otherwise. That is the rule `as.data.frame()` reports its bounds by, and it applies to the frame as a whole rather than to a row: the level names itself in the column headings and nowhere else, so a matrix holding a stored interval on one row and a rebuilt one on the next would report two kinds of interval under one heading. A stored pair need not be the one recomputing gives, since a bootstrap or profile interval is asymmetric about the estimate and even a normal one that was rounded on its way into the frame is a different number.

In the conditional reading the rows are the outcome model's coefficients, which parm names and indexes in the same two ways, and the limits are the normal ones built from the corrected covariance at every level, since the limits the result stores belong to the effects the marginal reading reports.

`coef()`, `vcov()`, and `confint()` raise an error of class `causalgenerics_invalid_argument_effects` when effects names neither reading, and when the result's own field does. They raise an error of class `causalgenerics_unsupported_reading_marginal` or `causalgenerics_unsupported_reading_conditional`, and of the general class `causalgenerics_unsupported_reading`, when effects names a reading the result records that it does not support.

`nobs()` returns a single integer, the number of observations the outcome model was fitted on, which is the number the estimates were computed from and not necessarily the number the weighting model saw.

`df.residual()` returns the residual degrees of freedom the fitted variance object reports, as an integer when that number is a whole one an integer can hold and as the double the object gave when it is not. A fractional count is what a penalized or smooth fit spends, and it comes back as it stands rather than truncated to a count the fit did not spend. `NA_integer_` comes back when the result records no fitted variance object or that object reports no residual degrees of freedom.

`weights()` returns the outcome model's weights as its model frame stores them, so a concrete weight class such as `psw` comes back as itself, or `NULL` when the outcome model was fitted unweighted.

`model.frame()` returns the outcome model's model frame, which is the data the reported estimates were computed from. The `(weights)` column a weighted frame carries is deliberately dropped, since tooling that reads the columns of a frame as the variables a model was fitted on, such as prediction and averaging packages, would otherwise treat the estimation weights as one of them; `weights()` is where those are reported. A model whose own method has no frame to give raises that model's error rather than one of this package's.

`estimand()` returns the estimand the result records, which is the one the weights the estimates were computed under targeted. There is deliberately no `estimand<-()` method for a result: assigning a new estimand would relabel those estimates rather than recompute them, so a caller who writes the assignment is told that no method exists.

The reading these methods report

`coef()`, `vcov()`, and `confint()` report the surface the result's effects field names, and take an effects argument that names one for a single call. The marginal reading is the causal contrast estimates, which is what these methods reported before the field existed and what a result that records no mode still reports. The conditional reading is the outcome model's coefficient surface: `coef()` gives the model's coefficients, `vcov()` gives their covariance, and `confint()` bounds them.

The covariance the conditional reading reports is the block of the joint estimation that the fitting package attached with `new_ipw_model()`, never the one the outcome model computed for itself. A model fitted as though its weights were fixed understates its uncertainty, because the weights were estimated from the same data, so there is nothing to fall back on when no corrected block is there. `vcov()` and `confint()` refuse the reading instead, and name which of the two ways the block can be missing they met. An outcome model that was never wrapped raises an error of class `causalgenerics_no_conditional_vcov`, which the package that produced the result answers by wrapping the outcome model with `new_ipw_model()` before it builds the result. An outcome model that carries the wrapper class with no covariance behind it raises an error of class `causalgenerics_no_vcov_ipw_model`, since a model that is already wrapped has nothing to gain from being wrapped again and the object itself is what is wrong. All of these carry the general class `causalgenerics_no_vcov`. `coef()` needs no such block and reports the coefficients either way.

The block and the coefficients are paired by name, and the block is reported in coefficient order whatever order it was attached in, so the variance read beside a coefficient is that coefficient's. A block whose labels cannot be paired with the coefficients raises an error of class `causalgenerics_conditional_vcov_mismatch`. A block of another size, one labeled with the parameter names of a stacked system, a model whose coefficients carry no names, and a model that reports no coefficients at all are the ways the pairing fails, and the message says which one it met. Reading such a block by position instead would report the covariance of other parameters under this model's coefficient names. A block that has to be reordered is refused for the same reason when a coefficient name is repeated: reordering by a name two coefficients share reports the first one's variance for both. A block already in coefficient order is reported as it was attached, since nothing is indexed and nothing can be misplaced.

A result records which of the two readings it supports, and naming the other one is refused rather than answered with a surface whose numbers are not the quantities the reading names. The refusal is the one `as_marginal()` and `as_conditional()` raise, of class `causalgenerics_unsupported_reading`, since a caller cares that the reading is not there rather than which surface asked for it. A result that supports one reading answers for that one exactly as a result supporting both does.

`nobs()`, `df.residual()`, `weights()`, `model.frame()`, and `estimand()` describe the fit rather than a surface of it, so they answer the same way in either reading and take no effects argument.

See Also

`new_ipw()` for the result class and the fields these methods read.

Examples

```
dat <- data.frame(
  x = rep(c(-1.5, -0.5, 0.5, 1.5), each = 5),
  z = rep(c(0, 1), 10),
  y = rep(c(0, 1, 1, 0, 1), 4)
)

ps <- fitted(glm(z ~ x, family = binomial(), data = dat))
wts <- ifelse(dat$z == 1, 1 / ps, 1 / (1 - ps))

# Written out literally, in the shape the `ipw()` return contract documents.
estimates <- data.frame(
  effect = c("rd", "log(rr)",
  estimate = c(0.199882, 0.560414),
  std.err = c(0.092425, 0.273519),
  z = c(2.1626, 2.0489),
  ci.lower = c(0.018732, 0.024326),
  ci.upper = c(0.381032, 1.096502),
  conf.level = 0.95,
  p.value = c(0.030570, 0.040470)
)

# The covariance of the two effects, which a method attaches to the estimates
# it returns. Both are computed from the same weighted means, so the
# off-diagonal entry is far from zero.
attr(estimates, "ipw_vcov") <- matrix(
  c(0.008542, 0.022753, 0.022753, 0.074813),
  nrow = 2,
  dimnames = list(estimates$effect, estimates$effect)
)

res <- new_ipw(
  estimand = "ate",
  wt_mod = glm(z ~ x, family = binomial(), data = dat),
  outcome_mod = glm(y ~ z, family = quasibinomial(), data = dat, weights = wts),
  estimates = estimates,
  se_method = "linearization",
  fit = NULL
)

coef(res)
vcov(res)

# At the stored level the stored limits come back.
confint(res)

# At any other level they are recomputed.
confint(res, parm = "rd", level = 0.9)
```

```

nobs(res)

# The linearization path records no fitted variance object.
df.residual(res)

head(weights(res))

# The data the estimates were computed from, without the `(weights)` column
# the fitting machinery stored beside the variables the formula named.
head(model.frame(res))

estimand(res)

# The conditional reading reports the outcome model's coefficient surface
# rather than the effects.
coef(res, effects = "conditional")

# Its covariance is the corrected block a fitting package attaches to the
# outcome model. This one carries none, and the covariance the model computed
# for itself is not a substitute: it treats the estimated weights as fixed.
try(vcov(res, effects = "conditional"))

outcome_mod <- glm(y ~ z, family = quasibinomial(), data = dat, weights = wts)

# Standing in for the outcome block of a stacked sandwich, which a method
# computes from the estimating equations of both steps.
corrected <- vcov(outcome_mod) * 1.4

conditional <- as_conditional(new_ipw(
  estimand = "ate",
  wt_mod = glm(z ~ x, family = binomial(), data = dat),
  outcome_mod = new_ipw_model(outcome_mod, corrected),
  estimates = estimates,
  se_method = "linearization",
  fit = NULL
))

# A result that records the conditional reading answers in it with nothing
# named at the call site.
coef(conditional)
vcov(conditional)
confint(conditional)

```

Description

The two readings of a result, defined for the class `new_ipw()` constructs and for the pooled results `pool_ipw()` returns.

- `as_marginal()` returns the result reporting the causal contrast estimates.
- `as_conditional()` returns the result presenting the outcome model's coefficient surface.

Usage

```
as_marginal(x, ...)
```

```
as_conditional(x, ...)
```

Arguments

<code>x</code>	An ipw or ipw_pooled object. These generics dispatch on this argument.
<code>...</code>	Arguments passed to methods.

Details

Both surfaces exist on an ipw result that supports both readings, so on one of those these generics record which one the result presents rather than computing anything. They set the `effects` field of the `new_ipw()` contract and read the `readings` field beside it, which makes them the supported way to move a result between the two readings: a caller writes `as_conditional(res)` rather than assigning to the field, and a result that does not support the reading asked for is refused rather than left recording one it has no surface for.

The methods on ipw hold for a result that supports both readings, which is what a result records unless the package that built it said otherwise. Asking such a result for either reading is always answerable: the methods never error on one, asking twice says what asking once said, and a result that goes out to the other reading and back is the result that went in.

A result that supports one reading is refused the other rather than moved to it. The `readings` field says which ones a result can present, and a fitting package records one of them when the other has no meaning for the analysis it ran. Asking for a reading outside that set raises an error of class `causalgenerics_unsupported_reading_marginal` or `causalgenerics_unsupported_reading_conditional`, and of the general class `causalgenerics_unsupported_reading`, which carries the reading asked for under `effects` and the set the result records under `readings`. The reading such a result does support is the no-op it is on a result carrying both.

A result built before the fields existed carries six or seven fields rather than eight, since the two were added one at a time. It reads as `marginal` where it records no mode, which is the mode every method produced then, and as `supporting both readings` where it records no set, which is what every result was assumed to support when none of them recorded otherwise.

The generics live here for the reason `print()` does. Two packages each registering `as_conditional.ipw()` would collide in the shared S3 method table, and a caller writing against a result would then get whichever package was installed last rather than the contract. There is no marginal or conditional reading of an object that is not an IPW result, so the default method signals an error rather than inventing one.

Value

x presenting the reading asked for. The methods on `ipw` change the `effects` field and nothing else, so every other field comes back as it went in, the covariance attached to `estimates` and the set of readings the result supports included. A result that does not support the reading asked for raises an error rather than returning one. The methods on `ipw_pooled` exchange `estimates`, `pooling`, and the recorded mode with the reading stored under `alternate`, so the reading that was presented is what the returned result stores there, and the components shared by both readings come back as they went in. A pooled result that does not carry the reading asked for raises an error rather than returning one.

The readings a pooled result carries

`pool_ipw()` pools both readings of one set of results whenever it can compute both, and stores the one the call did not name whole, under the `alternate` component. The methods on `ipw_pooled` therefore swap which reading the result presents rather than setting a field: the pooled `estimates`, the `pooling` diagnostics, and the recorded mode move together, and the components that describe the pooled analyses rather than a reading of them stay where they are.

Where both readings were pooled, the properties above hold. Asking a pooled result for the reading it already presents gives that result back, and a result taken out to the other reading and back is the result that went in. Totality is what these methods cannot keep. A reading the pooling could not compute is recorded as `unavailable` rather than `computed`, and asking for it raises an error of class `causalgenerics_pool_missing_surface_marginal` or `causalgenerics_pool_missing_surface_conditional`, and of the general class `causalgenerics_pool_missing_surface`. That condition carries the reading under `effects` and, under `reason`, the refusal that reading raised when it was pooled, which is the same wording the caller would have seen from `pool_ipw()` had they asked for it there.

A result pooled before both readings were kept records no `alternate` at all. Asking such a result for the other reading is refused with the same two classes and a reason of `NULL`, and the message says that pooling the results again gives a result carrying both.

See Also

`new_ipw()` for the result class and the field these generics set, and `pool_ipw()` for the pooled result and the reading it stores beside the one it presents.

Examples

```
dat <- data.frame(
  x = rep(c(-1.5, -0.5, 0.5, 1.5), each = 5),
  z = rep(c(0, 1), 10),
  y = rep(c(0, 1, 1, 0, 1), 4)
)

# Written out literally, in the shape the `ipw()` return contract documents.
estimates <- data.frame(
  effect = "rd",
  estimate = 0.199882,
  std.err = 0.092425,
  z = 2.1626,
  ci.lower = 0.018732,
```

```

    ci.upper = 0.381032,
    conf.level = 0.95,
    p.value = 0.030570
  )

  res <- new_ipw(
    estimand = "ate",
    wt_mod = glm(z ~ x, family = binomial(), data = dat),
    outcome_mod = glm(y ~ z, family = quasibinomial(), data = dat),
    estimates = estimates,
    se_method = "linearization",
    fit = NULL
  )

  # A method that names no mode reports marginal effects.
  res$effects

  as_conditional(res)$effects

  # Asking for the reading a result already reports gives the result back, and
  # the two readings round-trip.
  identical(as_marginal(res), res)
  identical(as_marginal(as_conditional(res)), res)

  # A result that supports one reading is refused the other rather than moved
  # to it.
  marginal_only <- new_ipw(
    estimand = "ate",
    wt_mod = glm(z ~ x, family = binomial(), data = dat),
    outcome_mod = glm(y ~ z, family = quasibinomial(), data = dat),
    estimates = estimates,
    se_method = "linearization",
    fit = NULL,
    readings = "marginal"
  )

  try(as_conditional(marginal_only))

  # Neither reading exists for an object that is not an IPW result.
  try(as_marginal(1:3))

```

Description

The methods the class `pool_ipw()` returns carries.

- `print()` summarizes the pooling and tabulates the pooled effects.

- `coef()` returns the pooled estimates.
- `vcov()` returns their pooled covariance.
- `confint()` returns their confidence limits.
- `nobs()` returns the smallest number of observations any pooled analysis was estimated from.
- `as.data.frame()` reports the pooled effects as a tidier-shaped table.
- `estimand()` returns the estimand the pooled analyses targeted.

Usage

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

## S3 method for class 'ipw_pooled'
coef(object, ..., effects = NULL)

## S3 method for class 'ipw_pooled'
vcov(object, ..., effects = NULL)

## S3 method for class 'ipw_pooled'
confint(object, parm, level = 0.95, ..., effects = NULL)

## S3 method for class 'ipw_pooled'
nobs(object, ...)

## S3 method for class 'ipw_pooled'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  ...,
  conf.int = FALSE,
  conf.level = NULL,
  exponentiate = FALSE,
  effects = NULL,
  call = sys.call()
)

## S3 method for class 'ipw_pooled'
estimand(x, ...)
```

Arguments

<code>x</code>	An <code>ipw_pooled</code> object.
<code>...</code>	Further arguments. These methods ignore them.
<code>object</code>	An <code>ipw_pooled</code> object.
<code>effects</code>	The reading to report, either "marginal" or "conditional". <code>NULL</code> , the default, reports the reading the result records; any other value overrides it for the one

	call and leaves the result as it is. A reading the pooling could not compute is refused with an error of class <code>causalgenerics_pool_missing_surface</code> , which is where <code>as_marginal()</code> and <code>as_conditional()</code> refuse it, and a result pooled before both readings were kept refuses a request for the other reading with the same classes, having nothing recorded to report.
<code>parm</code>	The rows to report an interval for, given either as the effect labels or as their positions. Missing means all of them.
<code>level</code>	The confidence level. At the level every row of the result stores, the stored limits are returned; at any other level, and for a frame whose rows do not agree on one, every row is rebuilt from <code>t</code> on its own degrees of freedom.
<code>row.names</code>	A character vector of row names for the returned table, or <code>NULL</code> for the automatic ones.
<code>optional</code>	Accepted for the <code>base::as.data.frame()</code> generic. Every column of the table is named, so there is nothing for it to make optional.
<code>conf.int</code>	If <code>TRUE</code> , append <code>conf.low</code> and <code>conf.high</code> columns after the rest of the table. Default is <code>FALSE</code> .
<code>conf.level</code>	The confidence level the bounds report. <code>NULL</code> , the default, uses the level the result records.
<code>exponentiate</code>	If <code>TRUE</code> , move the estimates that are on a log scale to their natural scale, as the section above describes. Default is <code>FALSE</code> .
<code>call</code>	The call to report a refusal against. A tidier that builds its table by calling <code>as.data.frame()</code> passes the call a user wrote, so the refusal names the function they typed rather than the delegation behind it. The default is the method's own call, which is what a direct call reports.

Details

These live here for the reason the `ipw` methods do. Two packages each registering `print.ipw_pooled()` would collide in the shared S3 method table, and a caller writing against a pooled result would then get whichever package was installed last rather than the contract.

Everything they read is a field `pool_ipw()` wrote. They recompute no pooling, so a result read back out of a saved file answers exactly as one just built does.

`ipw_pooled` deliberately does not inherit from `ipw`, and these methods are deliberately separate from the ones registered against that class. The two share column names and differ in what the numbers under them mean: a pooled result reports several analyses rather than one fit and refers its inference to `t` rather than to `z`, so a pooled result reaching `confint.ipw()` would come back with normal limits and nothing would say so.

Value

`print()` returns its input invisibly.

`coef()` returns a named numeric vector of pooled estimates, named by effect label in the marginal reading and by coefficient name in the conditional one.

`vcov()` returns the pooled covariance of those estimates, with the same labels as `dimnames` on both margins. A result whose analyses did not all carry a covariance records `none`, and raises an error of

class `causalgenerics_no_vcov_ipw_pooled`, and of the general class `causalgenerics_no_vcov`, rather than returning one built from the standard errors, which would report the effects as uncorrelated.

`confint()` returns a matrix with one row per effect `parm` selects, in the order `parm` gives them, and two columns holding the lower and upper limit, named by effect label and by the two tail probabilities as percentages. A character `parm` naming an effect the result does not report raises an error of class `causalgenerics_invalid_argument`.

`nobs()` returns a single integer.

`as.data.frame()` returns a plain data frame with the columns `term`, then `contrast` when the result names contrasts, then `group` when it names subgroups, then `estimate`, `std.error`, `statistic`, `df`, and `p.value`, with `conf.low` and `conf.high` appended when they are asked for. The pooled covariance travels on it under the `ipw_vcov` attribute unless the table was exponentiated.

`estimand()` returns the estimand the pooled results agreed on, which is the one the weights their estimates were computed under targeted.

The reading these methods report

A pooled result carries both readings whenever `pool_ipw()` could compute both: the one it presents and, under `alternate`, the one it does not. `coef()`, `vcov()`, `confint()`, and `as.data.frame()` report the reading the result presents, and take an `effects` argument naming one for a single call. Naming a reading reports it and leaves the result as it was; `as_marginal()` and `as_conditional()` are what change which reading a result presents.

The reading is settled before anything else the call asks for. `confint()` selects the rows `parm` names from the reading it was asked for, and `as.data.frame()` decides what `exponentiate = TRUE` moves and what it relabels from that reading too, so a conditional table is gated on the outcome model link whichever reading the result itself records. A reading the pooling could not compute is refused rather than answered with the one the result holds, and a value naming neither reading raises an error of class `causalgenerics_invalid_argument_effects`, and of the general class `causalgenerics_invalid_argument`, as it does everywhere else in this package.

`print()` reports the reading the result presents and takes no such argument. It names that reading in its heading, and a caller who wants the other one moves the result to it first. `nobs()` and `estimand()` describe the pooled analyses rather than a reading of them, so they answer the same way in either reading and take no `effects` argument.

The degrees of freedom

Every pooled effect carries its own degrees of freedom, in the `df` column of the estimates frame. They are the Barnard-Rubin adjusted count, which depends on how much of that effect's variance came from between the imputations, so two effects pooled from the same analyses ordinarily differ in them. `confint()` and `as.data.frame()` refer each row to `t` on its own count rather than to the normal or to a single count for the table; with few imputations those counts can be in single figures, where the difference is large.

There is deliberately no `df.residual()` method. Residual degrees of freedom are a property of one fit, and a pooled result has as many as it had imputations. The per-effect count above is not a residual count and differs from row to row, so it is reported in the column beside the interval and the p-value it was used for, and `df.residual()` on a pooled result finds no method and no field and gives `NULL`.

The confidence limits

`confint()` and `as.data.frame()` return the limits the result stores when every row of the frame records the level asked for, and rebuild every row otherwise. That is the rule both keep on an unpooled result, and it is a rule about the frame rather than about a row: a stored pair need not be the one recomputing gives, since it may have been rounded on its way into the frame, so a surface mixing stored limits with rebuilt ones would report two kinds of interval under one pair of column headings with nothing on it to say which is which.

`conf.level = NULL`, the `as.data.frame()` default, names the level the frame records, which is the level `pool_ipw()` built its limits at; a frame whose rows disagree records none, and the limits are rebuilt at 0.95.

Exponentiating

On a marginal table `exponentiate = TRUE` means what it means for an unpooled result. The rows labeled `log(rr)` and `log(or)` are matched exactly, their estimate and their bounds move to the natural scale, and the two terms are relabeled `rr` and `or`. The interval is settled before the scale is, so a rebuilt bound is a t half width on the log scale added to an estimate on the log scale and exponentiated afterwards. The standard error, the statistic, and the p-value describe the log scale and stay there, and the `ipw_vcov` attribute is dropped rather than carried, since it would describe neither the table it sits on nor anything else.

A conditional table has no rows labeled as ratios to pick out, so the link the outcome models were fitted with settles the question for the whole table: a `logit` link puts every coefficient on the log odds scale and a `log` link puts every coefficient on the log risk scale, and both are scales an exponential undoes. Every estimate moves and no term is relabeled, since the terms are coefficient names and a coefficient does not change its name with the scale its estimate is reported on. Every other link raises an error of class `causalgenerics_exponentiate_link`, and of the classes `causalgenerics_invalid_argument_exponentiate` and `causalgenerics_invalid_argument`, rather than exponentiating coefficients that describe nothing once exponentiated.

See Also

`pool_ipw()`, which produces these results, `as_marginal()` and `as_conditional()`, which move one between the readings it carries, and `new_ipw()` for the unpooled result they are pooled from.

Examples

```
dat <- data.frame(
  x = rep(c(-1.5, -0.5, 0.5, 1.5), each = 5),
  z = rep(c(0, 1), 10),
  y = rep(c(0, 1, 1, 0, 1), 4)
)

imputed <- lapply(1:3, function(i) {
  completed <- dat
  completed$y[i] <- 1 - completed$y[i]
  completed
})

estimate <- list(c(0.20, 0.55), c(0.30, 0.60), c(0.40, 0.65))
```

```

std_err <- list(c(0.10, 0.25), c(0.20, 0.30), c(0.30, 0.35))

fits <- Map(
  function(completed, estimate, std.err) {
    new_ipw(
      estimand = "ate",
      wt_mod = glm(z ~ x, family = binomial(), data = completed),
      outcome_mod = glm(y ~ z, family = quasibinomial(), data = completed),
      estimates = data.frame(
        effect = c("rd", "log(rr)"),
        estimate = estimate,
        std.err = std.err,
        z = estimate / std.err,
        ci.lower = estimate - 1.96 * std.err,
        ci.upper = estimate + 1.96 * std.err,
        conf.level = 0.95,
        p.value = 2 * pnorm(-abs(estimate / std.err))
      ),
      se_method = "linearization",
      fit = NULL
    )
  },
  imputed,
  estimate,
  std_err
)

pooled <- pool_ipw(fits)

pooled

coef(pooled)

# Wider than the normal limits would be, since the degrees of freedom are
# what few imputations leave.
confint(pooled)

nobs(pooled)

# The tidier-shaped table, with the degrees of freedom the statistic beside
# it is referred to.
as.data.frame(pooled)

# With an interval, and the ratio on its natural scale.
as.data.frame(pooled, conf.int = TRUE, exponentiate = TRUE)

# Residual degrees of freedom belong to one fit, so a pooled result has no
# method for them; the per-effect count is in the table above.
df.residual(pooled)

```

`joint-exposure-accessors`*Read the declaration a joint exposure carries*

Description

These three functions read back what `joint_exposure()` recorded, so that a package holding a column of exposures can act on the crossing without rebuilding it.

`is_joint_exposure()` reports whether an object declares a crossing. `joint_components()` reports which two treatments were crossed and the levels each of them takes. `joint_reference()` reports the cell the effects are reported against.

Usage

```
is_joint_exposure(x)
```

```
joint_components(x)
```

```
joint_reference(x)
```

Arguments

<code>x</code>	A joint exposure, as built by <code>joint_exposure()</code> . <code>is_joint_exposure()</code> accepts any object.
----------------	--

Details

They are plain functions rather than generics. The declaration lives in the attributes of one class, and a package that meets a joint exposure needs the same answer this package would give rather than a place to register a different one.

`joint_components()` gives the component names in the order the crossing was declared in, and each component's levels in the order the crossing varies them, which is the order that settles the cell labels. `joint_reference()` gives the first level of the vector, which is the cell crossing the two components' reference levels.

Value

`is_joint_exposure()` returns a single TRUE or FALSE. `joint_components()` returns a named list of two character vectors, one per component, named for the treatments. `joint_reference()` returns the reference cell's label as a single string.

See Also

`joint_exposure()`, which records what these read back.

Examples

```
x <- joint_exposure(
  qsmk = c(0, 1, 0, 1),
  exercise = c("no", "no", "yes", "yes")
)

is_joint_exposure(x)
joint_components(x)
joint_reference(x)
```

joint_exposure	<i>Declare an exposure that crosses two treatments</i>
----------------	--

Description

`joint_exposure()` crosses two discrete treatments into one categorical exposure and records the declaration on the result: which two treatments were crossed, in what order, which levels each of them takes, and which cell the effects are reported against. The package that builds the weights, the package that checks balance over the cells, and the package that reports the effects all read that one declaration, so they agree on which cells exist without any of them owning it.

Usage

```
joint_exposure(...)
```

Arguments

... Exactly two named vectors, one per treatment, recycled to a common length. Each may be a factor, a character vector, or a numeric or logical vector. The names are the treatment names the cell labels are written from, and the two must be distinct.

Details

Conditioning on a variable and jointly intervening on two treatments are different causal questions, and this class serves the second. Effect modification asks how the effect of one treatment differs across the levels of something else, which need not be a treatment and need not be anything an analysis could set; what comes back is one effect of that one treatment per subgroup, and a result reporting those carries the group column `new_ipw()` documents. Interaction asks what would happen under an intervention that sets both treatments at once; what comes back is one exposure with a cell per combination, reported against the reference cell. The packages that implement `ipw()` methods develop the distinction, in the weights they fit and in the effects they report. What is settled here is the declaration all of them read.

The result is a factor. Its class vector is `c("joint_exposure", "factor", "vctrs_vctr", "integer")`, with "factor" ahead of "vctrs_vctr" so that the formula machinery treats a joint exposure natively: `model.matrix()` gives exactly what a plain factor over the same cells gives, assign and contrasts attributes included.

The cells are the crossing of the two components' levels, labeled "name1 = level1, name2 = level2" and ordered with the first component varying fastest, which is the order `interaction()` produces. The reference cell crosses the two components' own reference levels, and that ordering is what puts it first, where the contrasts a model builds are read against it. A factor component's reference level is the first of its `levels()`; any other component's is the smallest of its values, which makes `0` the reference of a 0/1 treatment and `FALSE` the reference of a logical one.

Every cell of the crossing must be populated. A cell nobody falls in is a positivity violation rather than a small sample: the joint effect of the two treatments is not identified in those data, and no weight, contrast, or balance check over the crossing can be computed. Construction refuses it, as it refuses a component that never varies and a component with a missing value.

The two components must also be named distinctly. The names are what tell the treatments apart in the cells, so one name used twice would label every cell with the same variable on either side of it and leave nothing downstream able to tell which component a cell varies. Construction refuses that as well.

The declaration survives every operation that keeps the full set of cells, including `[, vctrs::vec_slice(), sort(),` and a round trip through a data frame, and it survives combining two joint exposures that declare the same crossing. Operations that narrow or rewrite the level set give it up and say so: `droplevels(), x[i, drop = TRUE],` and `levels<-` each warn and return a plain factor. So does combining a joint exposure with a factor, with a character vector, or with a joint exposure that declares a different crossing. Casting the other way is refused outright, because a crossing cannot be recovered from labels that merely look like one.

Two ways of combining escape all of that and give a bare factor without a word, because neither reaches a method this package can register. `unlist()` combines the underlying codes in base C code without dispatching at all, and `c()` dispatches on its first argument, so a combine that begins with a plain factor never reaches this class. Reach for `vctrs::vec_c()` where either could apply.

Value

A vector of class `c("joint_exposure", "factor", "vctrs_vctr", "integer")` with one level per cell of the crossing, carrying the component levels and the cell labels the accessors read back.

See Also

`is_joint_exposure()`, `joint_components()`, and `joint_reference()`, which read the declaration back off the vector.

Examples

```
x <- joint_exposure(
  qsmk = c(0, 1, 0, 1),
  exercise = c("no", "no", "yes", "yes")
)
x
```

```
levels(x)
joint_components(x)
joint_reference(x)
```

```
# The formula machinery sees the factor it is, and reports every cell
```

```
# against the reference one.
colnames(model.matrix(~x, data.frame(x = x)))
```

match_exposure_type	<i>Resolve the exposure type a function will work on</i>
---------------------	--

Description

match_exposure_type() turns the exposure_type argument a function exposes into the type it will actually work on. A caller either declares one of the three types or asks for "auto", and a function declares through valid_types which of them it can answer.

Usage

```
match_exposure_type(
  exposure_type = c("auto", "binary", "categorical", "continuous"),
  .exposure,
  valid_types = c("auto", "binary", "categorical", "continuous"),
  arg = ".exposure",
  announce = TRUE,
  call = rlang::caller_env()
)
```

Arguments

exposure_type	The type the caller declared, or "auto" to detect it from .exposure. A calling function passes its own argument through.
.exposure	The exposure vector to classify.
valid_types	The types the calling function can answer, including "auto" where it offers detection. The unsupported refusal names this set without "auto", which is a request for detection rather than a type an exposure can have.
arg	The name the announcement gives the exposure argument. A function whose exposure argument is named something else passes its own name, so the announcement never points at an argument the caller does not have.
announce	Should the detected type be announced? The announcement reports which branch the analysis took, so a caller passes FALSE where that is not worth reporting, such as one detecting several exposures at once. It is suppressed for every call, whatever this argument says, when options(causalgenerics.quiet = TRUE).
call	The environment both refusals report as their calling context.

Details

Resolution has two tiers, and which one a request falls into is what the refusal says. A type this package knows but the calling function does not support is refused as unsupported, with an error of class `causalgenerics_unsupported_exposure_type`, and the message names the types the function does support. A string that is no exposure type at all is refused as an unrecognized value instead, by `rlang::arg_match()`, which names the choices and suggests the nearest one. Both refusals report the environment given in call, so a consumer package names its own function rather than this one.

A detected type outside `valid_types` reaches the same unsupported refusal as a declared one. Naming a type and letting it be detected therefore give the same answer, and a caller cannot get further into a function by leaving the type off.

A declared type that the function supports is returned as it stands. The data is not consulted on that path and nothing is announced: the declaration is how the caller models the exposure, and a heuristic over unique values does not overrule it. A function that also wants a declaration the data cannot carry refused calls `check_forced_type()`, which is a separate decision.

A function that narrows `valid_types` narrows the default of its own `exposure_type` argument to match. The two are matched against each other, so a function offering all four choices while supporting three of them reports the mismatch on the first call that leaves the argument at its default.

Value

A single string: "binary", "categorical", or "continuous".

See Also

`detect_exposure_type()`, which supplies the reading under "auto", and `check_forced_type()`, which refuses a declaration the data cannot carry.

Examples

```
# A declared type is returned as it stands, without consulting the data.
match_exposure_type("continuous", c(0, 1, 2, 3))

# `"auto"` reads the exposure and announces what it read.
match_exposure_type("auto", c(0, 1, 1, 0))

# A function that works on two of the types says so.
try(match_exposure_type(
  "continuous",
  c(0, 1),
  valid_types = c("auto", "binary", "categorical")
))
```

new_causal_wts	<i>Construct a causal weight vector</i>
----------------	---

Description

`new_causal_wts()` is the low-level constructor for the abstract weight class that the r-causal ecosystem shares. It is intended for package developers building a concrete weight class, not for end users, and it assumes `x` is already a double vector.

Usage

```
new_causal_wts(x = double(), subclass, ...)
```

Arguments

<code>x</code>	A double vector of weights.
<code>subclass</code>	A single non-empty string naming the concrete weight class.
<code>...</code>	Metadata to store as attributes on the result. A NULL value leaves the attribute absent.

Details

Weights built here are `vctrs` vectors over a double, so the class vector is `c(subclass, "causal_wts", "vctrs_vctr", "double")`. Callers may rely on that shape and on its order: `subclass` first, so that methods for the concrete class take precedence, then `causal_wts`, so that the concrete class inherits every method registered against the abstract one. `subclass` must therefore be a single non-empty string. Anything else would change the length of the class vector or name a class that no method can be registered against.

The dots carry whatever metadata the concrete class records, such as the estimand or the exposure groups. This constructor neither names nor types those fields. A field passed as NULL leaves its attribute absent rather than recording a NULL, which is what lets a concrete constructor default its metadata to NULL and pass it straight through.

The abstract layer reaches well past construction. A concrete class inherits the metadata accessors `is_causal_wt()`, `estimand()`, and `estimand<-()`, and it inherits every read-only operation that names no metadata field and so gives the answer the underlying double would: `vec_math()`, the Summary group generic, `min()`, `max()`, `range()`, `median()`, `quantile()`, `summary()`, `anyDuplicated()`, `diff()`, `[]`, and the six comparison operators. `ess()` arrives as well, through a default that computes the Kish effective sample size for any numeric vector.

Do not write any of those again for a concrete class. A method registered on the subclass takes precedence over the shared one, and the comparison operators in particular do more than delegate: they short-circuit `vec_equal()` and `vec_compare()`, which would otherwise route an expression such as `weights > 0` through the concrete class's own `vec_ptype2()` method and signal whatever that method signals on a downgrade, once per call. `glm.fit()` evaluates comparisons of that shape repeatedly inside `profile.glm()`, so a single profiled confidence interval on a weighted fit would emit the same warning a hundred times over.

What a concrete class does still own is its `vec_ptype2()`, `vec_cast()`, `vec_ptype_abbr()`, `vec_ptype_full()`, `vec_restore()`, and `vec_arith()` methods, for three distinct reasons.

`vec_ptype2()` and `vec_cast()` cannot be inherited at all. `vctr`s resolves them through `s3_method_specific()`, which keys on `class(x)[[1]]` and never walks the rest of the class vector, so a `causal_wts` method is never found.

`vec_ptype_abbr()` and `vec_ptype_full()` are likewise not reached from the abstract class. A concrete class that omits either one falls back to the `vctr`s default and prints a bare class name where the estimand should appear.

`vec_restore()` and `vec_arith()` do dispatch through `causal_wts`, but they still have to be written for the concrete class, because they reconcile metadata that this package knows nothing about. An abstract `vec_restore()` that copies attributes wholesale errors on named weights and re-attaches index-typed metadata at the wrong length after slicing, and an abstract `vec_arith()` would silently legalize arithmetic between two different concrete weight classes.

Value

A vector of class `c(subclass, "causal_wts", "vctr", "double")`.

See Also

[estimand\(\)](#) and [is_causal_wt\(\)](#), the accessors this class supplies methods for, and [causal_wts_ptype2\(\)](#), the coercion rule a concrete class calls from its own `vec_ptype2()` method.

Examples

```
wt <- new_causal_wts(c(1.2, 0.8), subclass = "my_wts", estimand = "ate")
class(wt)
estimand(wt)
```

new_ipw

Construct an inverse probability weighted result

Description

`new_ipw()` is the low-level constructor for the object that every `ipw()` method returns. It is intended for package developers writing an `ipw()` method, not for end users, and it assumes its arguments are already validated.

Usage

```
new_ipw(
  estimand,
  wt_mod,
  outcome_mod,
  estimates,
  se_method,
  fit,
```

```

    effects = "marginal",
    readings = c("marginal", "conditional")
)

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

## S3 method for class 'ipw'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  ...,
  conf.int = FALSE,
  conf.level = 0.95,
  exponentiate = FALSE,
  effects = NULL,
  call = sys.call()
)

```

Arguments

<code>estimand</code>	The causal estimand the method targeted, such as "ate" or "att".
<code>wt_mod</code>	The weighting object: the fitted model that produced the weights.
<code>outcome_mod</code>	The fitted weighted outcome model.
<code>estimates</code>	A data frame of effect estimates, in the shape the return value describes. It has to be a data frame, and it has to name each row's effect measure in an effect column of strings or of factor levels, with a measure named in every row, since every label the result reports is read off that column. Anything else is refused with an error of class <code>causalgenerics_invalid_argument_estimates</code> , and of the general class <code>causalgenerics_invalid_argument</code> .
<code>se_method</code>	The standard error method that ran, such as "mestimation" or "linearization".
<code>fit</code>	The fitted variance object, or NULL when the method has none.
<code>effects</code>	For <code>new_ipw()</code> , the presentation mode the result reports its effects in, either "marginal" or "conditional"; a method that names no mode reports marginal effects. For <code>as.data.frame()</code> , the reading to report: NULL, the default, reports the reading the result records, and any other value overrides it for the one call and leaves the result as it is. A reading the result does not support is refused with an error of class <code>causalgenerics_unsupported_reading</code> .
<code>readings</code>	The readings the result supports, one or both of "marginal" and "conditional". The default is both, which is what every result supported before the field existed. A method whose analysis has no meaning under one of them names the other, and the mode the result records has to be one of the readings named.
<code>x</code>	An ipw object.
<code>...</code>	Further arguments. Neither method reads them.

row.names	A character vector of row names for the returned table, or NULL for the automatic ones.
optional	Accepted for the <code>base::as.data.frame()</code> generic. Every column of the table is named, so there is nothing for it to make optional.
conf.int	If TRUE, append <code>conf.low</code> and <code>conf.high</code> columns after the rest of the table. Default is FALSE.
conf.level	The confidence level the bounds report. At the level every row of estimates records, the stored bounds are returned; at any other level they are the normal approximation built from the estimate and its standard error. Default is 0.95.
exponentiate	If TRUE, move the estimates that are on a log scale to their natural scale. In the marginal reading those are the log risk ratio and the log odds ratio, and the two terms are relabeled "rr" and "or"; in the conditional reading the outcome model's link settles it for the whole table, as the section above describes. The confidence bounds move with the estimates. Standard errors, statistics, and p-values remain on the log scale, and the <code>ipw_vcov</code> attribute is dropped rather than carried, since it describes the estimates on the scale they were estimated on. Default is FALSE.
call	The call to report a refusal against. A tidier that builds its table by calling this method passes the call a user wrote, so the refusal names the function they typed rather than the delegation behind it. The default is the method's own call, which is what a direct call reports.

Details

The result layer is shared so that an IPW estimate reads the same way whichever package produced it. A package supplying an `ipw()` method builds its return here and inherits the `print()` and `as.data.frame()` methods registered against the class rather than writing its own. Two packages each defining `print.ipw()` would collide in the shared S3 method table, which is the situation this package exists to prevent.

The field names and their order are part of the contract, since callers read fields by name and print the object positionally. `fit` is present on every path, including the ones that have no fitted variance object to report, and `effects` and `readings` are present whether or not the method that built the result named a mode or a set of readings.

`print()` writes the estimand and the call of each component model, then the table of the surface the result's presentation mode names. The section below describes the two modes and what each one tabulates.

`as.data.frame()` reports the result as a tidier-shaped table rather than as a copy of the estimates component. In the marginal reading its columns are `term`, then `contrast` when the result names contrasts, then `group` when it names subgroups, then `estimate`, `std.error`, `statistic`, and `p.value`. Those are the names the tidier convention uses, so a fitting package's `tidy()` method is this table read as a tibble and nothing more. The estimates component itself is unchanged by any of what follows.

The conditional reading reports the outcome model's coefficients under those same headings, one row per coefficient, with `term` naming the coefficient and `std.error` the standard error the corrected covariance implies. It names no contrasts and no subgroups: those key the rows of the effects table, and a coefficient is named by itself. It is the table `as.data.frame()` reports for a

pooled result in the same reading, without the `df` column a pooled result has and an unpooled one does not.

A result whose outcome model carries no corrected block has no such standard error to report, and the reading is refused with an error of class `causalgenerics_no_conditional_vcov` rather than reported from the standard errors the model computed for itself. That is where the table parts company with `print()`, which writes those coefficients under a note saying that no covariance from the joint estimation is recorded: a printed table is read by someone who reads the note with it, and a data frame is read by code that would take the column for the corrected one.

`conf.int = TRUE` appends `conf.low` and `conf.high` after the other columns, and `conf.level` names the level they report. The level is an argument rather than a column, since a column would repeat one number down every row and be read as part of the table rather than as the level the two bounds beside it were built at. The bounds estimates stores are returned when every row of the frame records the level asked for. They need not be the normal pair: a bootstrap or profile interval is asymmetric about the estimate, and even a normal one rounded on its way into the frame is not the number recomputing gives. At any other level, and for a frame whose rows disagree about the level or record none, the bounds are the normal approximation built from the estimate and its standard error. The conditional reading has no stored pair to prefer, since the bounds the frame holds belong to the effects the other reading reports, so its bounds are the normal approximation at every level.

With `exponentiate = TRUE` the `log(rr)` and `log(or)` rows of the marginal table move to their natural scale, exponentiating the point estimate and the confidence bounds and relabeling the two terms "rr" and "or". Standard errors, statistics, and p-values stay on the log scale, where the inference is done, and the interval is settled before the scale is: bounds recomputed at another level are built on the log scale and exponentiated afterwards. The covariance described below travels on the returned table under the same `ipw_vcov` attribute while the rows are on the scale they were estimated on, and is dropped when `exponentiate = TRUE`, since a matrix left attached there would describe neither the table it sits on nor anything else.

A conditional table has no rows labeled as ratios to pick out, so the link the outcome model was fitted with settles the question for the whole table: a `logit` link puts every coefficient on the log odds scale and a `log` link puts every coefficient on the log risk scale, and both are scales an exponential undoes. Every estimate moves and no term is relabeled, since a coefficient name names the term rather than the scale its estimate is reported on. Every other link raises an error of class `causalgenerics_exponentiate_link`, and of the classes `causalgenerics_invalid_argument_exponentiate` and `causalgenerics_invalid_argument`, rather than exponentiating coefficients that describe nothing once exponentiated. That is the rule `as.data.frame()` keeps for a pooled result, in the same words.

Value

`new_ipw()` returns an S3 object of class `ipw`: a list of the following eight components, in this order.

`estimand` The causal estimand, such as "ate" or "att".

`wt_mod` The weighting object: the fitted model that produced the weights.

`outcome_mod` The fitted outcome model.

`estimates` A data frame with one row per effect measure and the following columns: `effect` (the measure name, as a character or factor column naming a measure in every row, which the constructor requires), `estimate` (point estimate), `std.err` (standard error), `z` (z-statistic),

`ci.lower` and `ci.upper` (confidence interval bounds), `conf.level`, and `p.value`. For a categorical exposure the data frame also has a `contrast` column, placed after `effect`, naming the non-reference level and reference level of each contrast. A frame stored against an earlier version of this contract names that column comparison. The older name is read as an alias for the canonical one wherever the column is read, so a result holding such a frame labels its rows and reports its table exactly as one holding a `contrast` column does. A method written now writes `contrast`. A result reported once per level of a grouping variable also has a `group` column, placed after the `contrast` column when one is present and after `effect` when the result names no contrasts, naming the subgroup each row was estimated in as a "var = value" string such as "sex = 0". That column has one spelling and no alias. Both optional columns are absent rather than constant when the result reports one contrast or one group, since a column repeating a single value down the table would read as a contrast or a subgroup that was named. `group` must be character and must name a subgroup in every row; the constructor refuses anything else with an error of class `causalgenerics_invalid_argument_estimates`, and of the general class `causalgenerics_invalid_argument`, since a label pasted from such a column would relabel every row it keys without failing anywhere.

`se_method` The standard error method used, such as "mestimation" or "linearization".

`fit` The fitted object the variance estimator produced, or NULL. A method that stacks estimating equations records the M-estimator here; the linearization path has no such object and records NULL.

`effects` The presentation mode, either "marginal" or "conditional". The marginal reading shows the causal contrast estimates and the conditional reading presents the outcome model's coefficient surface. See [as_marginal\(\)](#) and [as_conditional\(\)](#).

`readings` The readings the result supports, one or both of "marginal" and "conditional", and always including the mode above. A result stored before the field existed carries fewer components and is read as supporting both readings.

`print()` returns its input invisibly. `as.data.frame()` returns a plain data frame under the tidier column names described above, with the confidence bounds appended when they are asked for: the effect estimates in the marginal reading, and the outcome model's coefficients, one row each, in the conditional one.

The effect labels

Every row of estimates has a label, and it is the label rather than the position that `print()` writes down the side of its table and that `coef()`, `vcov()`, and `confint()` name their results with. The label is the columns that name a row, pasted together in the order they appear in the frame's contract: the effect column, then contrast when the result names contrasts, then group when it names subgroups. A binary exposure estimated for the whole sample is labeled "rd", a categorical one "rd b vs a", and one reported by subgroup "rd b vs a sex = 0". A categorical exposure repeats each effect measure across its contrasts and a subgroup analysis repeats each one across its groups, so effect alone would name several rows the same thing.

The presentation mode

A result reports its effects in one of two readings, recorded in the `effects` field. The "marginal" reading shows the causal contrast estimates the method targeted; the "conditional" reading presents

the outcome model's coefficient surface. The field says which one the result presents, and `as_marginal()` and `as_conditional()` are how a caller moves a result between them.

Which readings a result can present at all is the separate fact the readings field records. Both surfaces exist on most results and not on all of them: an exposure entering the outcome model through several columns has no single coefficient to read as the conditional effect, and a package that builds such a result records the reading it can answer for. A result records both unless the method that built it said otherwise, and the mode it records has to be one of them, so a result never presents a reading it does not support. Asking one for the reading it does not support raises an error of class `causalgenerics_unsupported_reading`, whether it is asked through the mode generics, through `coef()`, `vcov()`, or `confint()`, or through the effects argument of `as.data.frame()`. A result stored before the field existed records no set and is read as supporting both.

A printed result names its mode twice, since the two readings are different tables of different numbers: once on an `Effects:` line beside the estimand, and once in the heading of the table itself. The marginal reading tabulates the effect estimates the result stores, under `Marginal estimates:`. The conditional reading tabulates the outcome model's coefficients, under `Conditional estimates (outcome model):`, with the standard errors implied by the corrected covariance a fitting package attaches through `new_ipw_model()`. Which coefficient each entry of that block belongs to is what its labels say rather than its row order, so a block attached in another order still prints each standard error beside the coefficient it belongs to.

An outcome model a fitting package never wrapped is still printed: the coefficients are written on their own, followed by a note saying that no covariance from the joint estimation is recorded, rather than beside the standard errors the model computed for itself. An outcome model that carries the `new_ipw_model()` class with no covariance behind it is refused instead, with an error of class `causalgenerics_no_vcov_ipw_model`, and one whose block cannot be paired with its coefficients with an error of class `causalgenerics_conditional_vcov_mismatch`. The note answers a package that has not adopted the contract by telling the reader to wrap the model, and that advice has already been taken in both of those cases: the object is what is wrong, not the package that produced it.

An outcome model that reports no coefficients has no rows to tabulate under that heading, and the printed form says so in place of the table.

The covariance of the effects

A method that can compute the covariance of the effects it reports attaches it to the estimates data frame as an attribute named `ipw_vcov`. The value is a square numeric matrix whose row order is the row order of estimates and whose dimnames on both margins are the effect labels above. `vcov()` reads that attribute and raises an error when it is absent, so a method that has no covariance to report attaches none rather than a substitute: the standard errors in estimates give the diagonal of the matrix and say nothing about the off-diagonal entries, which are not zero for effects estimated from the same weighted means.

See Also

`ipw()`, the generic these results come from, and `as_marginal()` and `as_conditional()` for the presentation mode.

Examples

```

dat <- data.frame(
  x = rep(c(-1.5, -0.5, 0.5, 1.5), each = 5),
  z = rep(c(0, 1), 10),
  y = rep(c(0, 1, 1, 0, 1), 4)
)

# Written out literally, in the shape the return contract documents. These
# stand in for what an `ipw()` method would compute from the models below.
estimates <- data.frame(
  effect = c("rd", "log(rr)", "log(or)"),
  estimate = c(0.199882, 0.560414, 0.878313),
  std.err = c(0.092425, 0.273519, 0.418661),
  z = c(2.1626, 2.0489, 2.0979),
  ci.lower = c(0.018732, 0.024326, 0.057753),
  ci.upper = c(0.381032, 1.096502, 1.698873),
  conf.level = 0.95,
  p.value = c(0.030570, 0.040470, 0.035910)
)

res <- new_ipw(
  estimand = "ate",
  wt_mod = glm(z ~ x, family = binomial(), data = dat),
  outcome_mod = glm(y ~ z, family = quasibinomial(), data = dat),
  estimates = estimates,
  se_method = "linearization",
  fit = NULL
)

res

# The tidier-shaped table the result reports.
as.data.frame(res)

# With an interval, and the ratios on their natural scale.
as.data.frame(res, conf.int = TRUE, exponentiate = TRUE)

# A result whose analysis has no conditional reading records the one it
# supports, and the other is refused rather than reported.
marginal_only <- new_ipw(
  estimand = "ate",
  wt_mod = glm(z ~ x, family = binomial(), data = dat),
  outcome_mod = glm(y ~ z, family = quasibinomial(), data = dat),
  estimates = estimates,
  se_method = "linearization",
  fit = NULL,
  readings = "marginal"
)

try(as.data.frame(marginal_only, effects = "conditional"))

```

new_ipw_model

Attach a corrected covariance to a component model of an IPW fit

Description

`new_ipw_model()` wraps one of the models an `ipw()` method was given so that the model reports the covariance the two-step estimation implies rather than the one it computed for itself. It is intended for package developers writing an `ipw()` method, not for end users, and beyond the shape of `vcov` it assumes its arguments are already validated.

Usage

```
new_ipw_model(model, vcov)

## S3 method for class 'ipw_model'
vcov(object, ...)
```

Arguments

<code>model</code>	The fitted component model to wrap, such as the outcome model or the weighting model of an <code>ipw()</code> result.
<code>vcov</code>	The corrected covariance of that model's parameters: a square numeric matrix with dimnames on both margins.
<code>object</code>	An <code>ipw_model</code> object.
<code>...</code>	Further arguments. <code>vcov()</code> ignores them.

Details

A weighted outcome model fitted as though its weights were fixed understates its own uncertainty, because the weights were estimated from the same data. A method that solves the estimating equations of both steps as one stacked system has the corrected block on hand, and wrapping the model is how that block reaches a caller who writes `vcov(result$outcome_mod)`.

The wrapper is two changes and no more: `"ipw_model"` is prepended to the model's class vector, and the matrix is attached as the `ipw_vcov` attribute. Nothing is registered for the class beyond `vcov()`, so `predict()`, `coef()`, `model.frame()`, and every other generic walk past it to the model's own methods. That is what lets the corrected covariance travel with the model rather than beside it.

The dimnames of `vcov` are the fitting package's to choose. They are ordinarily `names(coef(model))`, but a method whose stacked system is parameterized some other way labels the block with its own parameter names. The constructor checks that both margins are labeled rather than what the labels say, since only the fitting package knows which block of the sandwich belongs to which model.

Value

`new_ipw_model()` returns `model` with `"ipw_model"` prepended to its class vector and `vcov` attached as the `ipw_vcov` attribute. Everything else about the model is untouched. `vcov()` returns that matrix, and raises an error of class `causalgenerics_no_vcov_ipw_model`, and of the general class

causalgenerics_no_vcov, when the attribute is absent. The constructor puts the class and the attribute on together, so a model carrying the class with nothing behind it passed through code that dropped its attributes and kept its class, and the covariance the model computed for itself is not a substitute for the one it lost.

See Also

[new_ipw\(\)](#) for the result these models are components of, and [ipw-accessors](#) for the accessors on the result itself.

Examples

```
dat <- data.frame(
  x = rep(c(-1.5, -0.5, 0.5, 1.5), each = 5),
  z = rep(c(0, 1), 10),
  y = rep(c(0, 1, 1, 0, 1), 4)
)

ps <- fitted(glm(z ~ x, family = binomial(), data = dat))
wts <- ifelse(dat$z == 1, 1 / ps, 1 / (1 - ps))
outcome_mod <- glm(y ~ z, family = quasibinomial(), data = dat, weights = wts)

# Standing in for the outcome block of a stacked sandwich. A real method
# computes this from the estimating equations of both steps; treating the
# weights as estimated rather than fixed ordinarily widens it.
corrected <- vcov(outcome_mod) * 1.4

wrapped <- new_ipw_model(outcome_mod, corrected)

class(wrapped)

# The corrected covariance rather than the model's own.
vcov(wrapped)

# Everything else about the model reaches its own methods.
coef(wrapped)
```

pool_ipw

Pool inverse probability weighted results over multiply imputed data

Description

`pool_ipw()` combines the [ipw\(\)](#) results fitted to each of a set of multiply imputed datasets into a single result, by Rubin's rules. It takes a plain list of results or the `mira` object `mice::with()` returns, and gives back an object of class `ipw_pooled`.

Usage

```
pool_ipw(fits, ..., effects = NULL, dfcom = NULL, conf_level = NULL)
```

Arguments

<code>fits</code>	The results to pool: a list of ipw objects, one per imputed dataset, or a <i>mira</i> . At least two are needed, since the between-imputation variance is estimated from the spread across them.
<code>...</code>	These dots exist so that every argument after <code>fits</code> is matched by name. Passing anything through them is an error.
<code>effects</code>	The reading the pooled result presents, either "marginal" or "conditional". NULL, the default, presents the reading the results record. Both readings are pooled whenever both can be, so this says which one the result reports and which one it stores beside it. The marginal reading pools the causal contrast estimates; the conditional reading pools the outcome models' coefficients, with the standard errors implied by the corrected covariance each one carries.
<code>dfcom</code>	The complete-data degrees of freedom. NULL, the default, reads them off the results as the section above describes.
<code>conf_level</code>	The level the pooled bounds report. NULL, the default, uses the level the results stored, or 0.95 when they store none.

Details

Each result is estimated on one completed dataset and carries a standard error that accounts for the weights having been estimated, and pooling those by Rubin's rules adds the uncertainty the imputation itself contributed: the order matters, since a standard error that treated the weights as fixed would be too small before the pooling ever saw it, and pooling the imputed datasets rather than the estimates would understate the uncertainty whatever the standard errors were.

For each effect the pooled estimate is the mean of the per-imputation estimates, and the pooled variance is the mean of their squared standard errors plus the between-imputation variance inflated by $1 + 1/m$ (Rubin, 1987). The degrees of freedom carry the Barnard-Rubin small-sample adjustment (Barnard and Rubin, 1999), which is why the complete-data count matters: with few imputations the pooled degrees of freedom can be far below what a single analysis reports, and the interval and the p-value are referred to t on them rather than to the normal.

mice is not a dependency, and nothing here is imported from it. A *mira* is recognized by its class and read through its `analyses` element, which is all the object is, so a package that produces one by another route is answered the same way.

Value

An S3 object of class `ipw_pooled`: a list of the following ten components, in this order.

<code>estimand</code>	The causal estimand every pooled result targeted.
<code>estimates</code>	A data frame with one row per effect and the following columns: <code>effect</code> (the measure name), <code>contrast</code> after it when the results name contrasts, <code>group</code> after that when they name subgroups, <code>estimate</code> (the pooled point estimate), <code>std.err</code> (the pooled standard error), <code>t</code> (the test statistic), <code>df</code> (the pooled degrees of freedom), <code>ci.lower</code> and <code>ci.upper</code> , <code>conf.level</code> , and <code>p.value</code> . The statistic and the p-value are referred to t on df rather than to the normal. When every pooled result carried the covariance of its effects, the pooled covariance is attached to this frame as the <code>ipw_vcov</code> attribute, with the effect labels as <code>dimnames</code> on both margins;

when any of them carried none, no attribute is attached, since a matrix built from a subset of the imputations would sit beside estimates built from all of them.

pooling A data frame keyed by the same columns that name a row of the frame above, holding `ubar` (the within-imputation variance), `b` (the between-imputation variance), `riv` (the relative increase in variance), `lambda` (the proportion of the total variance due to missingness), and `fmi` (the fraction of missing information).

se_method The standard error method every pooled result used.

effects The reading that was pooled, either "marginal" or "conditional".

m The number of results pooled.

dfcom The complete-data degrees of freedom the adjustment used.

nobs The smallest number of observations any pooled result was estimated from.

outcome_link The link every pooled result's outcome model was fitted with, which is the scale the effects are reported on.

alternate The reading the result does not present, in one of two shapes. When that reading was pooled it is a list of effects, naming which reading it is, and estimates and pooling, the two frames described above built for it and built the same way, the pooled covariance on the estimates frame included. When it could not be pooled it is a list of effects and reason, the message of the refusal that reading raised. `as_marginal()` and `as_conditional()` read this component to move the result between the two readings, and the accessors that take an `effects` argument read it the same way.

The two readings

Both readings of the results are pooled from one call whenever both can be pooled. `effects` says which of them the returned result presents, and the other is stored whole under `alternate`, so a caller moves a pooled result between the two afterwards with `as_marginal()` and `as_conditional()` rather than pooling again. Which reading was named therefore decides which pair of frames is the active one rather than what was computed at all: pooling one reading and moving to the other gives the frames pooling the other directly gives.

The reading the call names is pooled first and is not guarded, so a set that cannot be pooled on it is refused as it always was. The other reading is pooled under a guard, since a set that cannot be pooled on it is a result with one reading rather than a failed pooling: nothing about the reading the caller asked for is wrong. `alternate` then records that reading and the refusal it raised, and asking the result for it later raises that refusal in the words it used. The commonest case is a set of results whose outcome models carry no corrected covariance, which is what the conditional reading is pooled from.

The readings the results declare are read before either surface is, so a set whose results support one reading pools that one and no other. Naming the reading they do not support refuses the call with an error of class `causalgenerics_unsupported_reading_marginal` or `causalgenerics_unsupported_reading_conditional` in the words a result raises when the same reading is asked of it directly and a sentence saying which result of the set that was, since what the caller asked for is a reading of the analysis rather than a way of pooling it. Leaving `effects` at `NULL` pools the reading the results record and puts that same refusal on `alternate` as the reason the other reading has no frames, so `as_marginal()` or `as_conditional()` on the pooled result afterwards raises an error of class `causalgenerics_pool_missing_surface_marginal` or `causalgenerics_pool_missing_surface_conditional` carrying it. Every result has to support the reading being pooled, since the pooled estimate of an

effect is an average over all of them, so a set where one result alone lacks it is refused for that result. The caller has to go back to that one result, so the refusal names its position in `fits` in a sentence of its own and carries the position under `position`.

The components describing the analyses rather than a reading of them are shared by both readings. The estimand, the standard error method, the number of results, the complete-data degrees of freedom, the observation count, and the outcome model link are settled once, from the results themselves, and are what either reading reports.

What the results have to agree on

The pooled estimate of an effect is an average of the per-imputation ones, so the results have to be estimating the same things the same way. A set that disagrees about its estimand, its `se_method`, the presentation mode it records, the effects it reports, the confidence level it stores, or the link its outcome model was fitted with is refused with an error of class `causalgenerics_pool_mismatch`, and of a second class naming which of those it was. The differing values travel on the condition under `values`.

Those requirements are not all of one kind. The estimand, the standard error method, and the outcome model link describe the results rather than a reading of them, and so does the presentation mode when effects leaves it to be read; a disagreement about any of those refuses the call. Which effects a result reports and what level it reported them at are properties of one reading of it, and those requirements bind the reading being pooled: a disagreement confined to the reading the call did not name is recorded on `alternate` as the reason that reading could not be pooled rather than refused.

The effects have to agree as an ordered vector rather than as a set. The labels are what say which row is which, so two results reporting the same contrasts in different orders would otherwise have the `b vs a` rows of one averaged with the `c vs a` rows of the other.

A label carries the subgroup a row was estimated in along with the effect and the contrast, so a set whose results report different subgroups, or the same subgroups in different orders, disagrees about its labels and is refused through that requirement rather than through one of its own. A subgroup observed in one imputation and not in another is the case this answers: there is no row to average the missing one with, and pooling by position would combine two subgroups that answer different questions.

Two of those agreements are only required when the argument that would settle the question is left at `NULL`. Naming effects says which surface to pool and the stored mode is not read at all; naming `conf_level` says what the bounds report and the stored level is not read at all.

The complete-data degrees of freedom

`dfcom` is the number of observations the analysis had minus the number of parameters it fitted, before any data went missing. It is looked for in three places, in order:

1. The `dfcom` argument, when it is not `NULL`.
2. `df.residual()` on each result, which reports what the fitted variance object records, when at least one of them reports a number.
3. `df.residual()` on each result's outcome model, when at least one of those reports a number.

The smallest count is taken rather than the first, since the pooled inference is no stronger than the weakest analysis supports, and results that report nothing are passed over rather than making the

minimum missing. Wherever the count comes from it is at least 1, since the adjustment has nothing to work with at zero.

When nothing reports a count, `dfcom` is `Inf` and a warning of class `causalgenerics_pool_large_sample` says so. That is the honest answer for a large sample and the widest degrees of freedom the adjustment can give, which makes it the narrowest intervals, so it is said out loud rather than assumed quietly.

The scale the effects are pooled on

The estimates are pooled as they are stored. A result reports its ratio effects on the log scale, under the labels `log(rr)` and `log(or)`, which is the scale the inference is done on and the only one on which averaging the per-imputation estimates means anything. Nothing here exponentiates, and the pooled frame carries the same labels the results did. Moving the ratios to their natural scale is presentation rather than pooling, and belongs where the same choice is made for a single result.

References

Barnard, J. and Rubin, D. B. (1999). Small sample degrees of freedom with multiple imputation. *Biometrika*, 86(4), 948-955.

Rubin, D. B. (1987). *Multiple Imputation for Nonresponse in Surveys*. New York: John Wiley and Sons.

See Also

[new_ipw\(\)](#) for the results this pools and the fields it reads, and [as_marginal\(\)](#) and [as_conditional\(\)](#) for moving the pooled result between the two readings it carries.

Examples

```
dat <- data.frame(
  x = rep(c(-1.5, -0.5, 0.5, 1.5), each = 5),
  z = rep(c(0, 1), 10),
  y = rep(c(0, 1, 1, 0, 1), 4)
)

# Three imputations of that dataset, differing in one filled-in cell each.
imputed <- lapply(1:3, function(i) {
  completed <- dat
  completed$y[i] <- 1 - completed$y[i]
  completed
})

# The estimates each analysis produced, written out literally in the shape
# the `ipw()` return contract documents. What is being shown is the pooling,
# so these stand in for what a method would compute from the models beside
# them.
estimate <- list(c(0.20, 0.55), c(0.30, 0.60), c(0.40, 0.65))
std_err <- list(c(0.10, 0.25), c(0.20, 0.30), c(0.30, 0.35))

fits <- Map(
  function(completed, estimate, std.err) {
```

```

new_ipw(
  estimand = "ate",
  wt_mod = glm(z ~ x, family = binomial(), data = completed),
  outcome_mod = glm(y ~ z, family = quasibinomial(), data = completed),
  estimates = data.frame(
    effect = c("rd", "log(rr)"),
    estimate = estimate,
    std.err = std.err,
    z = estimate / std.err,
    ci.lower = estimate - 1.96 * std.err,
    ci.upper = estimate + 1.96 * std.err,
    conf.level = 0.95,
    p.value = 2 * pnorm(-abs(estimate / std.err))
  ),
  se_method = "linearization",
  fit = NULL
)
},
imputed,
estimate,
std_err
)

pooled <- pool_ipw(fits)

# The pooled effects. The standard errors are wider than the average of the
# per-imputation ones, which is the uncertainty the imputation contributed.
pooled$estimates

# How much of that uncertainty came from the imputation rather than the data.
pooled$pooling

# The complete-data count the adjustment used, read off the outcome models.
pooled$dfcom

# Naming one settles it directly, and the pooled degrees of freedom move.
pool_ipw(fits, dfcom = 500)$estimates$df

# The ratio effect stays on the log scale it was estimated on.
pooled$estimates$effect

# Both readings are pooled from one call. These outcome models carry no
# corrected covariance, which is what the conditional reading is pooled from,
# so that reading records why it has none rather than refusing the call.
pooled$alternate$effects

try(as_conditional(pooled))

```

Index

`as.data.frame()`, 35, 36
`as.data.frame.ipw(new_ipw)`, 33
`as.data.frame.ipw_pooled`
 (`ipw-pooled-methods`), 21
`as_conditional(ipw-modes)`, 18
`as_conditional()`, 14, 16, 23–25, 37, 38, 43, 45
`as_marginal(ipw-modes)`, 18
`as_marginal()`, 14, 16, 23–25, 37, 38, 43, 45

`base::as.data.frame()`, 23, 35

`causal-weights`, 2
`causal_wts_ptype2`, 4
`causal_wts_ptype2()`, 33
`check_forced_type`, 5
`check_forced_type()`, 8, 31
`coef()`, 37, 38
`coef.ipw(ipw-accessors)`, 13
`coef.ipw_pooled(ipw-pooled-methods)`, 21
`confint()`, 37, 38
`confint.ipw(ipw-accessors)`, 13
`confint.ipw_pooled`
 (`ipw-pooled-methods`), 21

`detect_exposure_type`, 7
`detect_exposure_type()`, 6, 10, 11, 31
`df.residual.ipw(ipw-accessors)`, 13

`ess`, 8
`ess()`, 32
`estimand(causal-weights)`, 2
`estimand()`, 4, 5, 32, 33
`estimand.ipw(ipw-accessors)`, 13
`estimand.ipw_pooled`
 (`ipw-pooled-methods`), 21
`estimand<-(causal-weights)`, 2
`exposure-type-helpers`, 10

`has_two_levels(exposure-type-helpers)`, 10

`interaction()`, 29
`ipw`, 11
`ipw()`, 3, 28, 33, 38, 40, 41
`ipw-accessors`, 13, 41
`ipw-modes`, 18
`ipw-pooled-methods`, 21
`is_categorical(exposure-type-helpers)`, 10
`is_causal_wt(causal-weights)`, 2
`is_causal_wt()`, 32, 33
`is_joint_exposure`
 (`joint-exposure-accessors`), 27
`is_joint_exposure()`, 29

`joint-exposure-accessors`, 27
`joint_components`
 (`joint-exposure-accessors`), 27
`joint_components()`, 29
`joint_exposure`, 28
`joint_exposure()`, 27
`joint_reference`
 (`joint-exposure-accessors`), 27
`joint_reference()`, 29

`match_exposure_type`, 30
`match_exposure_type()`, 6, 8
`model.frame.ipw(ipw-accessors)`, 13

`new_causal_wts`, 32
`new_causal_wts()`, 3, 5
`new_ipw`, 33
`new_ipw()`, 11–14, 17, 19, 20, 25, 28, 41, 45
`new_ipw_model`, 40
`new_ipw_model()`, 16, 38
`nobs.ipw(ipw-accessors)`, 13
`nobs.ipw_pooled(ipw-pooled-methods)`, 21

`observed_values`
 (`exposure-type-helpers`), 10

`pool_ipw`, 41

`pool_ipw()`, [19–21](#), [23–25](#)
`print.ipw(new_ipw)`, [33](#)
`print.ipw_pooled(ipw-pooled-methods)`,
 [21](#)

`rlang::arg_match()`, [31](#)

`vcov()`, [37](#), [38](#)
`vcov.ipw(ipw-accessors)`, [13](#)
`vcov.ipw_model(new_ipw_model)`, [40](#)
`vcov.ipw_pooled(ipw-pooled-methods)`, [21](#)

`weights.ipw(ipw-accessors)`, [13](#)