Package 'pintervals'

October 27, 2026

Type Package

Title Model Agnostic Prediction Intervals

Version 1.3.3

Description Provides tools for estimating model-agnostic prediction intervals using conformal prediction, bootstrapping, and parametric prediction intervals. The package is designed for ease of use, offering intuitive functions for both binned and full conformal prediction methods, as well as parametric interval estimation with diagnostic checks. Currently only working for continuous predictions. For details on the conformal and bin-conditional conformal prediction methods, see Randahl, Williams, and Hegre (2026) <DOI:10.1017/pan.2025.10010>.

License GPL (>= 3)

Encoding UTF-8

LazyData true

LazyDataCompression xz

Imports dplyr, foreach, Hmisc, MASS, purrr, Rcpp, stats, tibble

RoxygenNote 7.3.3

Depends R (>= 3.5)

LinkingTo Rcpp

Suggests knitr, rmarkdown, ranger, testthat, tidyr

VignetteBuilder knitr

NeedsCompilation yes

Author David Randahl [aut, cre],
Jonathan P. Williams [ctb],
Anders Hjort [ctb]

Maintainer David Randahl <david.randahl@fhs.se>

Repository CRAN

Date/Publication 2026-10-27 14:10:02 UTC

Contents

county_turnout . . . 2
county_turnout_panel . . . 4
elections . . . 8
interval_coverage . . . 9
interval_miscoverage . . . 10
interval_score . . . 12
interval_width . . . 14
pinterval_bccp . . . 15
pinterval_bootstrap . . . 19
pinterval_ccp . . . 22
pinterval_conformal . . . 27
pinterval_mondrian . . . 31
pinterval_parametric . . . 34

Index **38**

`county_turnout` *U.S. County-Level Turnout and Demographic Context (MIT Election Lab 2018 Election Analysis Dataset + Additions)*

Description

A county-level dataset (U.S.) with voter turnout and sociodemographic covariates.

Usage

```
data(county_turnout)
```

Format

A tibble with 3,107 rows and 22 variables:

state State name.

county County name.

fips County FIPS code.

turnout Observed turnout (proportion). Total presidential votes cast divided by the county Citizen Voting-Age Population (CVAP) from the 2012–2016 ACS special tabulation. (Earlier versions of this help page described the denominator as total population; that was incorrect – see NEWS for 1.3.0.)

total_population Total county population.

nonwhite_pct Percent non-white population.

foreignborn_pct Percent foreign-born population.

female_pct Percent female population.

age29andunder_pct Percent of population aged 29 or under.

age65andolder_pct Percent of population aged 65 or older.

median_hh_inc Median household income.

clf_unemploy_pct Percent unemployed in the civilian labor force.

lesscollege_pct Percent with less than college education.

lesshs_pct Percent with less than high school education.

rural_pct Percent rural.

ruralurban_cc Rural–urban continuum code.

predicted_turnout LOO-CV random-forest prediction of ‘turnout‘ (see Details).

division U.S. Census division.

region U.S. Census region.

geo_group Additional coarse geographic grouping variable (added).

longitude County centroid longitude (added).

latitude County centroid latitude (added).

Details

Superseded. This single-election (2016) cross-section is retained for backward compatibility and is still used by the package vignette and examples. For new work spanning multiple elections, prefer `county_turnout_panel`, a five-wave (2008–2024) county panel whose covariates are pulled directly from the U.S. Census Bureau API. `county_turnout_panel$turnout` uses the same votes / Citizen Voting-Age Population definition as `county_turnout$turnout` and reproduces its 2016 values for all but a handful of counties (see NEWS 1.3.0).

The dataset is based on the MIT Election Lab "2018 Election Analysis dataset" file, with four additions: (1) ‘turnout‘, calculated as the number of presidential votes cast divided by the Citizen Voting-Age Population, (2) ‘geo_group‘, a coarse geographic grouping variable for the counties, (3) county centroid coordinates (‘longitude‘, ‘latitude‘), and (4) ‘predicted_turnout‘. The variable ‘predicted_turnout‘ is generated using leave-one-out cross-validation (LOO-CV). For each county a random forest model is fit on the remaining counties with ‘turnout‘ as the outcome and all available *non-geographic* covariates as predictors. The fitted model is then used to predict turnout for the held-out county. Geographic features are excluded from the predictor set to avoid leaking spatial information into the prediction target. Concretely, identifiers and geographic variables (e.g., ‘state‘, ‘county‘, ‘fips‘, ‘division‘, ‘region‘, ‘geo_group‘, ‘longitude‘, ‘latitude‘) are excluded from the predictor set.

Below is example code (using ‘foreach‘) to reproduce ‘predicted_turnout‘. This is computationally expensive for LOO-CV; parallel execution is recommended.

library(dplyr) library(ranger) library(foreach) library(pintervals)

data("county_turnout", package = "pintervals")

Choose predictors: all numeric covariates except turnout + geographic/id vars model_data <- county_turnout %>% select(-c(state, county, fips, division, region, geo_group, longitude, latitude))

set.seed(101010) # The meaning of life in binary

pred_loo <- foreach(.i = seq_len(nrow(county_turnout)), .final = unlist) %do% {

train <- model_data[-.i, , drop = FALSE] test <- model_data[.i, , drop = FALSE]

```
fit <- ranger( formula = turnout ~ ., data = train )
predict(fit, data = test)$predictions[[1]]
}
county_turnout <- county_turnout %>% mutate(predicted_turnout = pred_loo)
```

Source

The base covariates originate from the MEDSL "2018 election context" file: https://github.com/MEDSL/2018-elections-unoffical/blob/master/election-context-2018.md. The variables ‘geo_group‘, ‘longitude‘, ‘latitude‘, and ‘predicted_turnout‘ are additions.

---

county_turnout_panel *U.S. County-Level Presidential Turnout and Demographic Context, 2008-2024 Panel*

Description

A tidy panel of U.S. county-level presidential-election voter turnout and sociodemographic context for the five presidential elections of 2008, 2012, 2016, 2020 and 2024. One row per county (county-equivalent) per election year. This is the multi-election generalisation of `county_turnout`; every covariate is re-derived directly from the U.S. Census Bureau API.

Usage

```
data(county_turnout_panel)
```

Format

A tibble with 15,543 rows and 38 variables:

state State name (or "District of Columbia").

county County / county-equivalent name (trailing " County"/" Parish" removed). Not unique on its own; pair with `fips`.

fips 5-character county FIPS code, stored as a character string with leading zeros (e.g. `"01001"`). (In `county_turnout` this column is numeric.)

election_year Presidential election year: 2008, 2012, 2016, 2020 or 2024. Panel key together with `fips`.

turnout Turnout rate = `total_votes / cvap`, using the same definition as `county_turnout$turnout`. Always ≤ 1 where present: 25 micro-county-years whose recorded ballots exceeded the noisy 5-year ACS CVAP estimate (turnout > 1) have been **dropped** from the panel. `NA` for 14 rows (Kalawao Co., HI in every wave; the 9 Connecticut planning regions in 2024) - these are **retained**.

total_votes Total presidential ballots cast (MEDSL), integer. Turnout numerator. `NA` on the 14 rows with no MEDSL match.

cvap Citizen Voting-Age Population (U.S. Census Bureau CVAP special tabulation, matched ACS 5-year window), integer. Turnout denominator.

votes_dem Presidential ballots for the Democratic candidate (MEDSL), integer.

votes_rep Presidential ballots for the Republican candidate (MEDSL), integer.

votes_other Presidential ballots for all other candidates combined - Libertarian, Green, other minor parties and recorded write-ins (MEDSL), integer. MEDSL only breaks out Libertarian/Green in some waves, so they are pooled here for a consistent panel.

dem_share `votes_dem / total_votes`, computed from the rounded integer counts.

rep_share `votes_rep / total_votes`.

other_share `votes_other / total_votes`. `dem_share + rep_share + other_share` is 1 up to rounding, except for 13 tiny 2024 counties (DC + Vermont towns) where upstream MEDSL sub-totals disagree by a few hundred ballots.

turnout_county_mean Per-election aggregate (constant within `election_year`): unweighted mean of county `turnout` across all counties with data for that wave (computed *before* the turnout > 1 rows are dropped, so it reflects the full electorate).

turnout_electorate Per-election aggregate: the "true" electorate-wide turnout for the wave = $\sum$ `total_votes` / $\sum$ `cvap` over all counties with data.

dem_share_county_mean Per-election aggregate: unweighted mean of county `dem_share`.

dem_share_electorate Per-election aggregate: electorate-wide Democratic share = $\sum$ `votes_dem` / $\sum$ `total_votes` (reproduces the national popular-vote share to a fraction of a point).

rep_share_county_mean Per-election aggregate: unweighted mean of county `rep_share`.

rep_share_electorate Per-election aggregate: electorate-wide Republican share.

other_share_county_mean Per-election aggregate: unweighted mean of county `other_share`.

other_share_electorate Per-election aggregate: electorate-wide other-candidate share.

total_population Total resident population (ACS table B01003), integer.

nonwhite_pct Percent not non-Hispanic White (from ACS B03002).

foreignborn_pct Percent foreign-born (ACS B05002).

female_pct Percent female (ACS B01001).

age29andunder_pct Percent aged 29 or younger (ACS B01001).

age65andolder_pct Percent aged 65 or older (ACS B01001).

median_hh_inc Median household income, inflation-adjusted to the ACS window's final year (ACS B19013), integer. `NA` for 2 Census-suppressed micro-county rows.

clf_unemploy_pct Unemployed as a percent of the civilian labor force (ACS B23025; B23001 for the 2008 wave).

lesscollege_pct Percent of the population 25+ with less than a bachelor's degree (ACS B15003; B15002 for the 2008 wave).

lesshs_pct Percent of the population 25+ with less than a high-school diploma (ACS B15003; B15002 for the 2008 wave).

rural_pct Percent of population classified rural by the nearest decennial census (table P2): 2010 census for the 2008/2012/2016 waves, 2020 census for the 2020/2024 waves. Held fixed across waves that share a census. `NA` for the 9 Connecticut planning regions in 2024.

ruralurban_cc USDA ERS Rural-Urban Continuum Code (integer 1-9; lower = more metropolitan). Vintage reused across adjacent elections: 2003 -> 2008; 2013 -> 2012 & 2016; 2023 -> 2020 & 2024.

division U.S. Census Bureau division (factor, 9 levels). State-level; inherited from `county_turnout`.

region U.S. Census Bureau region (factor, 4 levels). State-level; inherited from `county_turnout`.

geo_group Coarse `pintervals`-specific state grouping (factor, 15 levels). Inherited from `county_turnout`.

longitude County internal-point (approximate centroid) longitude, decimal degrees (Census Gazetteer).

latitude County internal-point latitude, decimal degrees (Census Gazetteer).

Details

Each covariate is matched to the ACS 5-year vintage whose window ends nearest the election: 2006-2010 (2008), 2008-2012 (2012), 2012-2016 (2016), 2016-2020 (2020), 2020-2024 (2024).

Coverage. 50 states + DC. **Alaska is excluded** - MEDSL reports Alaska by state-house district, not by borough, so no borough-level presidential returns exist (the original `county_turnout` also omits Alaska). Island territories are excluded (no presidential electors).

Votes by party. `votes_dem`, `votes_rep` and `votes_other` are the MEDSL county presidential returns collapsed to three groups. MEDSL breaks 2008-2016 into Democrat / Republican / Other only; 2020 adds Green and Libertarian and 2024 adds Libertarian, all of which are folded into `votes_other` so the split is consistent across waves. Where a county reports by voting mode with no total row, the mode rows are summed; administrative "TOTAL VOTES CAST" / "UNDERVOTES" style rows carried in the 2024 file are dropped. The three columns sum to `total_votes` exactly for 2008-2020 and for all but 13 tiny 2024 counties.

Per-election aggregates. The eight `*_county_mean` / `*_electorate` columns are constant within an `election_year` and are computed over every county with data for that wave, *before* the turnout > 1 rows are dropped. `*_county_mean` is the unweighted mean across counties; `*_electorate` is the pooled ratio ($\sum$ numerator / $\sum$ denominator) - the "true" per-election turnout and vote shares for the whole electorate.

ACS covariates are pulled directly from the U.S. Census Bureau API, not via IPUMS NHGIS. NHGIS's terms of use prohibit redistributing extracted data outside narrow journal exceptions, so ACS data obtained through NHGIS cannot be bundled in a CRAN package; the identical ACS detailed tables from the Census Bureau's own API are a U.S. Government work in the public domain. The reconstruction was validated against `county_turnout` (which is built from an NHGIS-sourced file): for the 2012-2016 ACS vintage all ten shared covariates match to floating-point tolerance across all 3,107 shared counties.

Geography / FIPS harmonisation. Geography is kept native to each wave (the matched ACS county list is the spine). Documented handling of known changes in the window:

- *Shannon County, SD (46113) -> Oglala Lakota County, SD (46102), 2015*: sources still using 46113 (MEDSL, RUCC 2013, 2010 census) are recoded to 46102 for the 2016/2020/2024 waves; 2008/2012 use 46113.
- *Bedford (independent) city, VA (51515) -> Bedford County, VA (51019), 2013*: separate row in 2008/2012 (native); folded into 51019 for 2016; 51019 only thereafter.
- *Valdez-Cordova Census Area, AK split (2019)* and *Wade Hampton -> Kusilvak, AK (2015)*: Alaska is excluded, so these are noted only for completeness.

- *Connecticut: 8 counties -> 9 planning regions (2022-vintage products)*: kept native per wave - the 2008-2020 waves use the 8 legacy counties (09001-09015), the 2024 wave uses the 9 planning regions (09110-09190). Connecticut is therefore not a consistent 5-wave panel: 2024 `turnout` and `rural_pct` are `NA` for the planning regions, and CT's 2020 `ruralurban_cc` is taken from RUCC 2013 (legacy counties).
- *Kansas City, MO* reports as its own unit (FIPS 2938000; mis-coded 36000 in the 2024 MEDSL file); its ballots - total and by party - are redistributed to Cass/Clay/Jackson/Platte counties by each county's share of the Kansas City place population from the nearest decennial census.

Native geography per wave is 3114 / 3114 / 3113 / 3113 / 3114 county-equivalents for 2008 / 2012 / 2016 / 2020 / 2024 (the Bedford (2016) and Connecticut (2024) changes above); after dropping the 6 / 3 / 4 / 6 / 6 county-years with turnout > 1, the shipped per-year row counts are 3108 / 3111 / 3109 / 3107 / 3108.

Rurality-code vintage caveat. USDA ERS revises the Rural-Urban Continuum roughly once per decade, so `ruralurban_cc` is shared by the 2012 & 2016 waves (2013 codes) and by the 2020 & 2024 waves (2023 codes).

A full variable-by-variable codebook is installed with the package at `system.file("extdata", "county_turnout_panel_codebook.md", package = "pintervals")`. The dataset is rebuilt end-to-end by `data-raw/county_turnout_panel.R`; see `data-raw/README.md` for the exact upstream citations and the regeneration procedure.

Source

- **Turnout numerator and votes by party**: MEDSL, "County Presidential Election Returns 2000-2024", Harvard Dataverse, doi:10.7910/DVN/VOQCHQ. License: **CC0 1.0**.
- **Turnout denominator**: U.S. Census Bureau, Citizen Voting-Age Population (CVAP) Special Tabulation from the ACS, https://www.census.gov/programs-surveys/decennial-census/about/voting-rights/cvap.html. **Public domain** (U.S. Government work).
- **Demographic covariates**: U.S. Census Bureau, American Community Survey 5-year estimates (detailed tables B01001, B01003, B03002, B05002, B15002/B15003, B19013, B23001/B23025), obtained directly from the Census Bureau API (https://www.census.gov/data/developers/data-sets/acs-5year.html). **Public domain**. Deliberately not obtained via IPUMS NHGIS, to comply with NHGIS's non-redistribution terms.
- **rural_pct**: U.S. Census Bureau, decennial census table P2 (Urban and Rural), 2010 and 2020, via the Census Bureau API. **Public domain**.
- **ruralurban_cc**: USDA Economic Research Service, Rural-Urban Continuum Codes (2003, 2013, 2023 releases), https://www.ers.usda.gov/data-products/rural-urban-continuum-codes/. **Public domain** (U.S. Government work).
- **longitude / latitude**: U.S. Census Bureau Gazetteer Files, https://www.census.gov/geographies/reference-files/time-series/geo/gazetteer-files.html. **Public domain**.
- **division / region / geo_group**: inherited verbatim from `pintervals::county_turnout` (U.S. Census Bureau regions and divisions; `geo_group` is a `pintervals` addition).

See Also

`county_turnout` for the original single-election cross-section.

`elections` *Election-Year Democracy Indicators from V-Dem (1946–2024)*

Description

A sample of election years from the V-Dem dataset covering 2,680 country-years between 1946 and 2024. Includes a range of democracy indices and related variables measured during years in which national elections were held.

Usage

```
elections
```

Format

´elections´ A tibble with 2,680 rows and 21 variables:

country_name Country name

year Election year

v2x_polyarchy Electoral democracy index

v2x_libdem Liberal democracy index

v2x_partipdem Participatory democracy index

v2x_delibdem Deliberative democracy index

v2x_egaldem Egalitarian democracy index

v2xel_frefair Free and fair elections index

v2x_frassoc_thick Freedom of association index

v2x_elecoff Elected officials index

v2eltrnout Voter turnout (V-Dem)

v2x_accountability Accountability index

v2xps_party Party system institutionalization

v2x_civlib Civil liberties index

v2x_corr Control of corruption index

v2x_rule Rule of law index

v2x_neopat Neo-patrimonial rule index

v2x_suffr Suffrage index

turnout Turnout percentage (external source)

hog_lost Factor indicating if head of government lost election

hog_lost_num Numeric version of `hog_lost`

Source

Data derived from the Varieties of Democracy (V-Dem) dataset, version 15, filtered to election years between 1946 and 2024. <https://www.v-dem.net/data/the-v-dem-dataset/>

`interval_coverage` *Empirical Coverage of Prediction Intervals*

Description

Calculates the mean empirical coverage rate of prediction intervals, i.e., the proportion of true values that fall within their corresponding prediction intervals.

Usage

```
interval_coverage(
  truth,
  lower_bound = NULL,
  upper_bound = NULL,
  intervals = NULL,
  return_vector = FALSE,
  na.rm = FALSE
)
```

Arguments

`truth`	A numeric vector of true outcome values.
`lower_bound`	A numeric vector of lower bounds of the prediction intervals.
`upper_bound`	A numeric vector of upper bounds of the prediction intervals.
`intervals`	Alternative input for prediction intervals as a list-column, where each element is a list with components 'lower_bound' and 'upper_bound'. Useful with non-contigous intervals, for instance constructed using the bin conditional conformal method wich can yield multiple intervals per prediction. See details.
`return_vector`	Logical, whether to return the coverage vector (TRUE) or the mean coverage (FALSE). Default is FALSE.
`na.rm`	Logical, whether to remove NA values before calculation. Default is FALSE.

Details

If the 'intervals' argument is provided, it should be a list-column where each element is a list containing 'lower_bound' and 'upper_bound' vectors. This allows for the calculation of coverage for non-contiguous intervals, such as those produced by certain conformal prediction methods such as the bin conditional conformal method. In this case, coverage is determined by checking if the true value falls within any of the specified intervals for each observation. If the user has some observations with contiguous intervals and others with non-contiguous intervals, they can provide both 'lower_bound' and 'upper_bound' vectors along with the 'intervals' list-column. The function will compute coverage accordingly for each observation based on the available information.

Value

A single numeric value between 0 and 1 representing the proportion of covered values.

Examples

```
library(dplyr)
library(tibble)

# Simulate example data
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rnorm(1000, mean = x1 + x2, sd = 1)
df <- tibble(x1, x2, y)

# Split into training, calibration, and test sets
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model on the log-scale
mod <- lm(y ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- predict(mod, newdata = df_cal)
pred_test <- predict(mod, newdata = df_test)

# Estimate normal prediction intervals from calibration data
intervals <- pinterval_parametric(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  dist = "norm",
  alpha = 0.1
)

# Calculate empirical coverage
interval_coverage(truth = df_test$y,
         lower_bound = intervals$lower_bound,
         upper_bound = intervals$upper_bound)
```

---

interval_miscoverage *Empirical Miscoverage of Prediction Intervals*

Description

Calculates the empirical miscoverage rate of prediction intervals, i.e., the difference between proportion of true values that fall within their corresponding prediction intervals and the nominal coverage rate (1 - alpha).

Usage

```
interval_miscoverage(truth, lower_bound, upper_bound, alpha, na.rm = FALSE)
```

Arguments

truth	A numeric vector of true outcome values.
lower_bound	A numeric vector of lower bounds of the prediction intervals.
upper_bound	A numeric vector of upper bounds of the prediction intervals.
alpha	The nominal miscoverage rate (e.g., 0.1 for 90% prediction intervals).
na.rm	Logical, whether to remove NA values before calculation. Default is FALSE.

Value

A single numeric value between -1 and 1 representing the empirical miscoverage rate. A value close to 0 indicates that the prediction intervals are well-calibrated.

Examples

```
library(dplyr)
library(tibble)

# Simulate example data
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rnorm(1000, mean = x1 + x2, sd = 1)
df <- tibble(x1, x2, y)

# Split into training, calibration, and test sets
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model on the log-scale
mod <- lm(y ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- predict(mod, newdata = df_cal)
pred_test <- predict(mod, newdata = df_test)

# Estimate normal prediction intervals from calibration data
intervals <- pinterval_parametric(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  dist = "norm",
  alpha = 0.1
)

# Calculate empirical coverage
interval_miscoverage(truth = df_test$y,
         lower_bound = intervals$lower_bound,
         upper_bound = intervals$upper_bound,
         alpha = 0.1)
```

`interval_score` *Mean Interval Score (MIS) for Prediction Intervals*

Description

Computes the mean interval score, a proper scoring rule that penalizes both the width of prediction intervals and any lack of coverage. Lower values indicate better interval quality.

Usage

```
interval_score(
  truth,
  lower_bound = NULL,
  upper_bound = NULL,
  intervals = NULL,
  return_vector = FALSE,
  alpha,
  na.rm = FALSE
)
```

Arguments

`truth`	A numeric vector of true outcome values.
`lower_bound`	A numeric vector of lower bounds of the prediction intervals.
`upper_bound`	A numeric vector of upper bounds of the prediction intervals.
`intervals`	Alternative input for prediction intervals as a list-column, where each element is a list with components 'lower_bound' and 'upper_bound'. Useful with non-contigous intervals, for instance constructed using the bin conditional conformal method wich can yield multiple intervals per prediction. See details.
`return_vector`	Logical, whether to return the interval score vector (TRUE) or the mean interval score (FALSE). Default is FALSE.
`alpha`	The nominal miscoverage rate (e.g., 0.1 for 90% prediction intervals).
`na.rm`	Logical, whether to remove NA values before calculation. Default is FALSE.

Details

The mean interval score (MIS) is defined as:

$$MIS = (ub - lb) + \frac{2}{\alpha}(lb - y) \cdot 1_{y<lb} + \frac{2}{\alpha}(y - ub) \cdot 1_{y>ub}$$

where \(y \) is the true value, and \([lb, ub] \) is the prediction interval.

If the 'intervals' argument is provided, it should be a list-column where each element is a list containing 'lower_bound' and 'upper_bound' vectors. This allows for the calculation of coverage for non-contiguous intervals, such as those produced by certain conformal prediction methods such as the bin conditional conformal method. In this case, coverage is determined by checking if the

true value falls within any of the specified intervals for each observation. If the user has some observations with contiguous intervals and others with non-contiguous intervals, they can provide both ‘lower_bound‘ and ‘upper_bound‘ vectors along with the ‘intervals‘ list-column. The function will compute coverage accordingly for each observation based on the available information.

Value

A single numeric value representing the mean interval score across all observations.

Examples

```
library(dplyr)
library(tibble)

# Simulate example data
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rnorm(1000, mean = x1 + x2, sd = 1)
df <- tibble(x1, x2, y)

# Split into training, calibration, and test sets
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model on the log-scale
mod <- lm(y ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- predict(mod, newdata = df_cal)
pred_test <- predict(mod, newdata = df_test)

# Estimate normal prediction intervals from calibration data
intervals <- pinterval_parametric(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  dist = "norm",
  alpha = 0.1
)

# Calculate empirical coverage
interval_score(truth = df_test$y,
         lower_bound = intervals$lower_bound,
         upper_bound = intervals$upper_bound,
         alpha = 0.1)
```

`interval_width` *Mean Width of Prediction Intervals*

Description

Computes the mean width of prediction intervals, defined as the average difference between upper and lower bounds.

Usage

```
interval_width(
  lower_bound = NULL,
  upper_bound = NULL,
  intervals = NULL,
  return_vector = FALSE,
  na.rm = FALSE
)
```

Arguments

`lower_bound`	A numeric vector of lower bounds of the prediction intervals.
`upper_bound`	A numeric vector of upper bounds of the prediction intervals.
`intervals`	Alternative input for prediction intervals as a list-column, where each element is a list with components 'lower_bound' and 'upper_bound'. Useful with non-contigous intervals, for instance constructed using the bin conditional conformal method wich can yield multiple intervals per prediction. See details.
`return_vector`	Logical, whether to return the width vector (TRUE) or the mean width (FALSE). Default is FALSE.
`na.rm`	Logical, whether to remove NA values before calculation. Default is FALSE.

Details

The mean width is calculated as:

$$\text{Mean Width} = \frac{1}{n} \sum_{i=1}^{n} (ub_i - lb_i)$$

where \(ub_i \) and \(lb_i \) are the upper and lower bounds of the prediction interval for observation \(i \), and \(n \) is the total number of observations.

If the 'intervals' argument is provided, it should be a list-column where each element is a list containing 'lower_bound' and 'upper_bound' vectors. This allows for the calculation of coverage for non-contiguous intervals, such as those produced by certain conformal prediction methods such as the bin conditional conformal method. In this case, coverage is determined by checking if the true value falls within any of the specified intervals for each observation. If the user has some observations with contiguous intervals and others with non-contiguous intervals, they can provide both 'lower_bound' and 'upper_bound' vectors along with the 'intervals' list-column. The function will compute coverage accordingly for each observation based on the available information.

Value

A single numeric value representing the mean width of the prediction intervals.

Examples

```
library(dplyr)
library(tibble)

# Simulate example data
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rnorm(1000, mean = x1 + x2, sd = 1)
df <- tibble(x1, x2, y)

# Split into training, calibration, and test sets
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model on the log-scale
mod <- lm(y ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- predict(mod, newdata = df_cal)
pred_test <- predict(mod, newdata = df_test)

# Estimate normal prediction intervals from calibration data
intervals <- pinterval_parametric(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  dist = "norm",
  alpha = 0.1
)

# Calculate empirical coverage
interval_width(lower_bound = intervals$lower_bound,
         upper_bound = intervals$upper_bound)
```

---

pinterval_bccp *Bin-Conditional Conformal Prediction Intervals for Continuous Predictions*

Description

This function calculates bin-conditional conformal prediction intervals with a confidence level of 1-alpha for a vector of (continuous) predicted values using inductive conformal prediction on a bin-by-bin basis. The intervals are computed using a calibration set with predicted and true values and their

associated bins. The function returns a tibble containing the predicted values along with the lower and upper bounds of the prediction intervals. Bin-conditional conformal prediction intervals are useful when the prediction error is not constant across the range of predicted values and ensures that the coverage is (approximately) correct for each bin under the assumption that the non-conformity scores are exchangeable within each bin.

Usage

```
pinterval_bccp(
  pred,
  calib = NULL,
  calib_truth = NULL,
  calib_bins = NULL,
  breaks = NULL,
  right = TRUE,
  contiguize = FALSE,
  alpha = 0.1,
  ncs_type = c("absolute_error", "relative_error", "za_relative_error",
    "heterogeneous_error", "raw_error"),
  grid_size = 10000,
  resolution = NULL,
  distance_weighted_cp = FALSE,
  distance_features_calib = NULL,
  distance_features_pred = NULL,
  normalize_distance = "none",
  distance_type = c("mahalanobis", "euclidean"),
 weight_function = c("gaussian_kernel", "caucy_kernel", "logistic", "reciprocal_linear")
)
```

Arguments

pred	Vector of predicted values
calib	A numeric vector of predicted values in the calibration partition, or a 2 column tibble or matrix with the first column being the predicted values and the second column being the truth values. If calib is a numeric vector, calib_truth must be provided.
calib_truth	A numeric vector of true values in the calibration partition. Only required if calib is a numeric vector
calib_bins	A vector of bin identifiers for the calibration set. Not used if breaks are provided.
breaks	A vector of break points for the bins to manually define the bins. If NULL, lower and upper bounds of the bins are calculated as the minimum and maximum values of each bin in the calibration set. Must be provided if calib_bins is not provided, either as a vector or as the last column of a calib tibble.
right	Logical, if TRUE the bins are right-closed (a,b] and if FALSE the bins are left-closed '[a,b)'. Only used if breaks are provided.
contiguize	Logical indicating whether to contiguize the intervals. TRUE will consider all bins for each prediction using the lower and upper endpoints as interval limits

to avoid non-contiguous intervals. FALSE will allows for non-contiguous intervals. TRUE guarantees at least appropriate coverage in each bin, but may suffer from over-coverage in certain bins. FALSE will have appropriate coverage in each bin but may have non-contiguous intervals. Default is FALSE.

`alpha` The confidence level for the prediction intervals. Must be a single numeric value between 0 and 1

`ncs_type` A string specifying the type of nonconformity score to use. Available options are:

- `"absolute_error"`: $|y - \hat{y}|$
- `"relative_error"`: $|y - \hat{y}|/\hat{y}$
- `"zero_adjusted_relative_error"`: $|y - \hat{y}|/(\hat{y} + 1)$
- `"heterogeneous_error"`: $|y - \hat{y}|/\sigma_{\hat{y}}$ absolute error divided by a measure of heteroskedasticity, computed as the predicted value from a linear model of the absolute error on the predicted values
- `"raw_error"`: the signed error $y - \hat{y}$

The default is `"absolute_error"`.

`grid_size` The number of points to use in the grid search between the lower and upper bound. Default is 10,000. A larger grid size increases the resolution of the prediction intervals but also increases computation time.

`resolution` Alternatively to grid_size. The minimum step size between grid points. Useful if the a specific resolution is desired. Default is NULL.

`distance_weighted_cp`

Logical. If `TRUE`, weighted conformal prediction is performed where the nonconformity scores are weighted based on the distance between calibration and prediction points in feature space. Default is `FALSE`. See details for more information.

`distance_features_calib`

A matrix, data frame, or numeric vector of features from which to compute distances when `distance_weighted_cp = TRUE`. This should contain the feature values for the calibration set. Must have the same number of rows as the calibration set. Can be the predicted values themselves, or any other features which give a meaningful distance measure.

`distance_features_pred`

A matrix, data frame, or numeric vector of feature values for the prediction set. Must be the same features as specified in `distance_features_calib`. Required if `distance_weighted_cp = TRUE`.

`normalize_distance`

Either 'minmax', 'sd', or 'none'. Indicates if and how to normalize the distances when distance_weighted_cp is TRUE. Normalization helps ensure that distances are on a comparable scale across features. Default is 'none'.

`distance_type` The type of distance metric to use when computing distances between calibration and prediction points. Options are 'mahalanobis' (default) and 'euclidean'.

`weight_function`

A character string specifying the weighting kernel to use for distance-weighted conformal prediction. Options are:

- `"gaussian_kernel"`: $w(d) = e^{-d^2}$
- `"caucy_kernel"`: $w(d) = 1/(1 + d^2)$
- `"logistic"`: $w(d) = 1//(1 + e^d)$
- `"reciprocal_linear"`: $w(d) = 1/(1 + d)$

The default is `"gaussian_kernel"`. Distances are computed as the Euclidean distance between the calibration and prediction feature vectors.

Details

'pinterval_bccp()' extends [pinterval_conformal()] to the bin-conditional setting, where prediction intervals are calibrated separately within user-specified bins. It is particularly useful when prediction error varies across the range of predicted values, as it enables locally valid coverage by ensuring that the coverage level $1 - \alpha$ holds within each bin—assuming exchangeability of non-conformity scores within bins.

For a detailed description of non-conformity scores, distance weighting and the general inductive conformal framework, see [pinterval_conformal()].

For 'pinterval_bccp()', the calibration set must include predicted values, true values, and corresponding bin identifiers or breaks for the bins. These can be provided either as separate vectors ('calib', 'calib_truth', and 'calib_bins' or 'breaks').

Bins endpoints can be defined manually via the 'breaks' argument or inferred from the calibration data. If 'contiguize = TRUE', the function ensures the resulting prediction intervals are contiguous across bins, potentially increasing coverage beyond the nominal level in some bins. If 'contiguize = FALSE', the function may produce non-contiguous intervals, which are more efficient but may be harder to interpret.

Value

A tibble with the predicted values and the lower and upper bounds of the prediction intervals. If contiguize = FALSE, the intervals may consist of multiple disjoint segments; in this case, the tibble will contain a list-column with all segments for each prediction.

See Also

pinterval_conformal

Examples

```
# Generate example data
library(dplyr)
library(tibble)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rlnorm(1000, meanlog = x1 + x2, sdlog = 0.5)

# Create bins based on quantiles
 bin <- cut(y, breaks = quantile(y, probs = seq(0, 1, 1/4)),
 include.lowest = TRUE, labels =FALSE)
df <- tibble(x1, x2, y, bin)
df_train <- df %>% slice(1:500)
```

```
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model to the training data
mod <- lm(log(y) ~ x1 + x2, data=df_train)

# Generate predictions on the original y scale for the calibration data
calib <- exp(predict(mod, newdata=df_cal))
calib_truth <- df_cal$y
calib_bins <- df_cal$bin

# Generate predictions for the test data

pred_test <- exp(predict(mod, newdata=df_test))

# Calculate bin-conditional conformal prediction intervals
pinterval_bccp(pred = pred_test,
calib = calib,
calib_truth = calib_truth,
calib_bins = calib_bins,
alpha = 0.1)
```

`pinterval_bootstrap` *Bootstrap Prediction Intervals*

Description

This function computes bootstrapped prediction intervals with a confidence level of 1-alpha for a vector of (continuous) predicted values using bootstrapped prediction errors. The prediction errors to bootstrap from are computed using either a calibration set with predicted and true values or a set of pre-computed prediction errors from a calibration dataset or other data which the model was not trained on (e.g. OOB errors from a model using bagging). The function returns a tibble containing the predicted values along with the lower and upper bounds of the prediction intervals.

Usage

```
pinterval_bootstrap(
  pred,
  calib,
  calib_truth = NULL,
  error_type = c("raw", "absolute"),
  alpha = 0.1,
  n_bootstraps = 1000,
  distance_weighted_bootstrap = FALSE,
  distance_features_calib = NULL,
  distance_features_pred = NULL,
  distance_type = c("mahalanobis", "euclidean"),
  normalize_distance = "none",
```

```
  weight_function = c("gaussian_kernel", "caucy_kernel", "logistic", "reciprocal_linear")
)
```

Arguments

pred	Vector of predicted values
calib	A numeric vector of predicted values in the calibration partition, or a 2 column tibble or matrix with the first column being the predicted values and the second column being the truth values. If calib is a numeric vector, calib_truth must be provided.
calib_truth	A numeric vector of true values in the calibration partition. Only required if calib is a numeric vector
error_type	The type of error to use for the prediction intervals. Can be 'raw' or 'absolute'. If 'raw', bootstrapping will be done on the raw prediction errors. If 'absolute', bootstrapping will be done on the absolute prediction errors with random signs. Default is 'raw'
alpha	The confidence level for the prediction intervals. Must be a single numeric value between 0 and 1
n_bootstraps	The number of bootstraps to perform. Default is 1000
distance_weighted_bootstrap	Logical. If TRUE, the function will use distance-weighted bootstrapping. Default is FALSE. If TRUE, the probability of selecting a prediction error is weighted by the distance to the predicted value using the specified distance function and weight function. If FALSE, standard bootstrapping is performed.
distance_features_calib	A matrix, data frame, or numeric vector of features from which to compute distances when `distance_weighted_cp = TRUE`. This should contain the feature values for the calibration set. Must have the same number of rows as the calibration set. Can be the predicted values themselves, or any other features which give a meaningful distance measure.
distance_features_pred	A matrix, data frame, or numeric vector of feature values for the prediction set. Must be the same features as specified in `distance_features_calib`. Required if `distance_weighted_cp = TRUE`.
distance_type	The type of distance metric to use when computing distances between calibration and prediction points. Options are 'mahalanobis' (default) and 'euclidean'.
normalize_distance	Either 'minmax', 'sd', or 'none'. Indicates if and how to normalize the distances when distance_weighted_cp is TRUE. Normalization helps ensure that distances are on a comparable scale across features. Default is 'none'.
weight_function	A character string specifying the weighting kernel to use for distance-weighted conformal prediction. Options are: • `"gaussian_kernel"`: $w(d) = e^{-d^2}$ • `"caucy_kernel"`: $w(d) = 1/(1 + d^2)$

- `"logistic"`: $w(d) = 1//(1 + e^d)$
- `"reciprocal_linear"`: $w(d) = 1/(1 + d)$

The default is `"gaussian_kernel"`. Distances are computed as the Euclidean distance between the calibration and prediction feature vectors.

Details

This function estimates prediction intervals using bootstrapped prediction errors derived from a calibration set. It supports both standard and distance-weighted bootstrapping. The calibration set must consist of predicted values and corresponding true values, either provided as separate vectors or as a two-column tibble or matrix. Alternatively, users may provide a vector of precomputed prediction errors if model predictions and truths are already processed.

Two types of error can be used for bootstrapping: - ‘"raw"‘: bootstrapping is performed on the raw signed prediction errors (truth - prediction), allowing for asymmetric prediction intervals. - ‘"absolute"‘: bootstrapping is done on the absolute errors, and random signs are applied when constructing intervals. This results in (approximately) symmetric intervals around the prediction.

Distance-weighted bootstrapping (‘distance_weighted_bootstrap = TRUE‘) can be used to give more weight to calibration errors closer to each test prediction. Distances are computed between the feature matrices or vectors supplied via ‘distance_features_calib‘ and ‘distance_features_pred‘. These distances are then transformed into weights using the selected kernel in ‘weight_function‘, with rapidly decaying kernels (e.g., Gaussian) emphasizing strong locality and slower decays (e.g., reciprocal or Cauchy) providing smoother influence. Distances can be geographic coordinates, predicted values, or any other relevant features that capture similarity in the context of the prediction task. The distance metric is specified via ‘distance_type‘, with options for Mahalanobis or Euclidean distance. The default is Mahalanobis distance, which accounts for correlations between features. Normalization of distances can be applied using the ‘normalize_distance‘ parameter. Normalization is primarily useful for euclidean distances to ensure that features on different scales do not disproportionately influence the distance calculations.

The number of bootstrap samples is controlled via the ‘n_bootstraps‘ parameter. For computational efficiency, this can be reduced at the cost of interval precision.

Value

A tibble with the predicted values, lower bounds, and upper bounds of the prediction intervals

Examples

```
library(dplyr)
library(tibble)

# Simulate some data
set.seed(42)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rlnorm(1000, meanlog = x1 + x2, sdlog = 0.4)
df <- tibble(x1, x2, y)

# Split into train/calibration/test
df_train <- df[1:500, ]
```

```
df_cal <- df[501:750, ]
df_test <- df[751:1000, ]

# Fit a log-linear model
model <- lm(log(y) ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- exp(predict(model, newdata = df_cal))
pred_test <- exp(predict(model, newdata = df_test))

# Compute bootstrap prediction intervals
intervals <- pinterval_bootstrap(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  error_type = "raw",
  alpha = 0.1,
  n_bootstraps = 1000
)
```

`pinterval_ccp` *Clustered Conformal Prediction Intervals for Continuous Predictions*

Description

This function computes conformal prediction intervals with a confidence level of $1 - \alpha$ by first grouping Mondrian classes into data-driven clusters based on the distribution of their nonconformity scores. The resulting clusters are used as strata for computing class-conditional (Mondrian-style) conformal prediction intervals. This approach improves local validity and statistical efficiency when there are many small or similar classes with overlapping prediction behavior. The coverage level $1 - \alpha$ is approximate within each cluster, assuming exchangeability of nonconformity scores within clusters.

The method supports additional features such as prediction calibration, distance-weighted conformal scores, and clustering optimization via internal validity measures (e.g., Calinski-Harabasz index or minimum cluster size heuristics).

Usage

```
pinterval_ccp(
  pred,
  pred_class = NULL,
  calib = NULL,
  calib_truth = NULL,
  calib_class = NULL,
  lower_bound = NULL,
  upper_bound = NULL,
  alpha = 0.1,
```

```
  ncs_type = c("absolute_error", "relative_error", "za_relative_error",
    "heterogeneous_error", "raw_error"),
  grid_size = 10000,
  resolution = NULL,
  n_clusters = NULL,
  cluster_method = c("kmeans", "ks"),
  cluster_train_fraction = 0.5,
  optimize_n_clusters = TRUE,
  optimize_n_clusters_method = c("calinhara", "min_cluster_size"),
  min_cluster_size = 150,
  min_n_clusters = 2,
  max_n_clusters = NULL,
  distance_weighted_cp = FALSE,
  distance_features_calib = NULL,
  distance_features_pred = NULL,
  distance_type = c("mahalanobis", "euclidean"),
  normalize_distance = "none",
 weight_function = c("gaussian_kernel", "caucy_kernel", "logistic", "reciprocal_linear")
)
```

Arguments

pred	Vector of predicted values
pred_class	A vector of class identifiers for the predicted values. This is used to group the predictions by class for Mondrian conformal prediction.
calib	A numeric vector of predicted values in the calibration partition, or a 2 column tibble or matrix with the first column being the predicted values and the second column being the truth values. If calib is a numeric vector, calib_truth must be provided.
calib_truth	A numeric vector of true values in the calibration partition. Only required if calib is a numeric vector
calib_class	A vector of class identifiers for the calibration set.
lower_bound	Optional minimum value for the prediction intervals. If not provided, the minimum (true) value of the calibration partition will be used. Primarily useful when the possible outcome values are outside the range of values observed in the calibration set. If not provided, the minimum (true) value of the calibration partition will be used.
upper_bound	Optional maximum value for the prediction intervals. If not provided, the maximum (true) value of the calibration partition will be used. Primarily useful when the possible outcome values are outside the range of values observed in the calibration set. If not provided, the maximum (true) value of the calibration partition will be used.
alpha	The confidence level for the prediction intervals. Must be a single numeric value between 0 and 1
ncs_type	A string specifying the type of nonconformity score to use. Available options are:

- "absolute_error": $|y - \hat{y}|$
- "relative_error": $|y - \hat{y}|/\hat{y}$
- "zero_adjusted_relative_error": $|y - \hat{y}|/(\hat{y} + 1)$
- "heterogeneous_error": $|y - \hat{y}|/\sigma_{\hat{y}}$ absolute error divided by a measure of heteroskedasticity, computed as the predicted value from a linear model of the absolute error on the predicted values
- "raw_error": the signed error $y - \hat{y}$

The default is "absolute_error".

grid_size The number of points to use in the grid search between the lower and upper bound. Default is 10,000. A larger grid size increases the resolution of the prediction intervals but also increases computation time.

resolution Alternatively to grid_size. The minimum step size between grid points. Useful if the a specific resolution is desired. Default is NULL.

n_clusters Number of clusters to use when combining Mondrian classes. Required if optimize_n_clusters = FALSE.

cluster_method Clustering method used to group Mondrian classes. Options are "kmeans" or "ks" (Kolmogorov-Smirnov). Default is "kmeans".

cluster_train_fraction
Fraction of the calibration data used to estimate nonconformity scores and compute clustering. Default is 1, which uses the entire calibration set for both clustering and interval estimation. See details for more discussion.

optimize_n_clusters
Logical. If TRUE, the number of clusters is chosen automatically based on internal clustering criteria.

optimize_n_clusters_method
Method used for cluster optimization. One of "calinhara" (Calinski-Harabasz index) or "min_cluster_size". Default is "calinhara".

min_cluster_size
Minimum number of calibration points per cluster. Used only when optimize_n_clusters_method = "min_cluster_size".

min_n_clusters Minimum number of clusters to consider when optimizing.

max_n_clusters Maximum number of clusters to consider. If NULL, the upper limit is set to the number of unique Mondrian classes minus 1.

distance_weighted_cp
Logical. If TRUE, weighted conformal prediction is performed where the nonconformity scores are weighted based on the distance between calibration and prediction points in feature space. Default is FALSE. See details for more information.

distance_features_calib
A matrix, data frame, or numeric vector of features from which to compute distances when distance_weighted_cp = TRUE. This should contain the feature values for the calibration set. Must have the same number of rows as the calibration set. Can be the predicted values themselves, or any other features which give a meaningful distance measure.

`distance_features_pred`
: A matrix, data frame, or numeric vector of feature values for the prediction set. Must be the same features as specified in `distance_features_calib`. Required if `distance_weighted_cp = TRUE`.

`distance_type`
: The type of distance metric to use when computing distances between calibration and prediction points. Options are 'mahalanobis' (default) and 'euclidean'.

`normalize_distance`
: Either 'minmax', 'sd', or 'none'. Indicates if and how to normalize the distances when distance_weighted_cp is TRUE. Normalization helps ensure that distances are on a comparable scale across features. Default is 'none'.

`weight_function`
: A character string specifying the weighting kernel to use for distance-weighted conformal prediction. Options are:

  - `"gaussian_kernel"`: $w(d) = e^{-d^2}$
  - `"caucy_kernel"`: $w(d) = 1/(1 + d^2)$
  - `"logistic"`: $w(d) = 1//(1 + e^d)$
  - `"reciprocal_linear"`: $w(d) = 1/(1 + d)$

  The default is `"gaussian_kernel"`. Distances are computed as the Euclidean distance between the calibration and prediction feature vectors.

Details

'pinterval_ccp()' builds on [pinterval_mondrian()] by introducing a clustered conformal prediction framework. Instead of requiring a separate calibration distribution for every Mondrian class, which may lead to unstable or noisy intervals when there are many small groups, the method groups similar Mondrian classes into clusters with similar nonconformity score distributions. Classes with similar prediction-error behavior are assigned to the same cluster. Each resulting cluster is then treated as a stratum for standard inductive conformal prediction.

Users may specify the number of clusters directly using the 'n_clusters' argument or optimize the number of clusters using the Calinski–Harabasz index or minimum cluster size heuristics.

Clustering can be computed using all calibration data or a subsample defined by 'cluster_train_fraction'. By default, the entire calibration set is used for both clustering and interval estimation, which may lead to overfitting. Setting 'cluster_train_fraction' to a value less than 1 (e.g., 0.5) can help mitigate this risk by using separate data for clustering and interval estimation, at the cost of potentially less stable cluster assignments with smaller calibration subsets. If data is limited, using the full calibration set for clustering may still be preferable, but users should be aware of the potential for overfitting and optimistic coverage estimates in this case.

Clustering is based on either k-means or Kolmogorov-Smirnov distance between nonconformity score distributions of the Mondrian classes, selected via the 'cluster_method' argument.

For a detailed description of non-conformity scores, distance-weighting, and the general conformal prediction framework, see [pinterval_conformal()], and for a description of Mondrian conformal prediction, see [pinterval_mondrian()].

Value

A tibble with predicted values, lower and upper prediction interval bounds, class labels, and assigned cluster labels. Attributes include clustering diagnostics (e.g., cluster assignments, coverage gaps, internal validity scores).

See Also

pinterval_conformal, pinterval_mondrian

Examples

```
library(dplyr)
library(tibble)

# Simulate data with 6 Mondrian classes forming 3 natural clusters
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
class_raw <- sample(1:6, size = 1000, replace = TRUE)

# Construct 3 latent clusters: (1,2), (3,4), (5,6)
mu <- ifelse(class_raw %in% c(1, 2), 1 + x1 + x2,
      ifelse(class_raw %in% c(3, 4), 2 + x1 + x2,
                                3 + x1 + x2))

sds <- ifelse(class_raw %in% c(1, 2), 0.5,
      ifelse(class_raw %in% c(3, 4), 0.3,
                        0.4))

y <- rlnorm(1000, meanlog = mu, sdlog = sds)

df <- tibble(x1, x2, class = factor(class_raw), y)

# Split into training, calibration, and test sets
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit model (on log-scale)
mod <- lm(log(y) ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- exp(predict(mod, newdata = df_cal))
pred_test <- exp(predict(mod, newdata = df_test))

# Apply clustered conformal prediction
intervals <- pinterval_ccp(
  pred = pred_test,
  pred_class = df_test$class,
  calib = pred_cal,
  calib_truth = df_cal$y,
  calib_class = df_cal$class,
```

```
  alpha = 0.1,
  ncs_type = "absolute_error",
  optimize_n_clusters = TRUE,
  optimize_n_clusters_method = "calinhara",
  min_n_clusters = 2,
  max_n_clusters = 4
)

# View clustered prediction intervals
head(intervals)
```

---

pinterval_conformal *Conformal Prediction Intervals of Continuous Values*

Description

This function calculates conformal prediction intervals with a confidence level of 1-alpha for a vector of (continuous) predicted values using inductive conformal prediction. The intervals are computed using either a calibration set with predicted and true values or a set of pre-computed non-conformity scores from the calibration set. The function returns a tibble containing the predicted values along with the lower and upper bounds of the prediction intervals.

Usage

```
pinterval_conformal(
  pred,
  calib = NULL,
  calib_truth = NULL,
  alpha = 0.1,
  ncs_type = c("absolute_error", "relative_error", "za_relative_error",
    "heterogeneous_error", "raw_error"),
  lower_bound = NULL,
  upper_bound = NULL,
  grid_size = 10000,
  resolution = NULL,
  distance_weighted_cp = FALSE,
  distance_features_calib = NULL,
  distance_features_pred = NULL,
  distance_type = c("mahalanobis", "euclidean"),
  normalize_distance = "none",
 weight_function = c("gaussian_kernel", "caucy_kernel", "logistic", "reciprocal_linear")
)
```

Arguments

pred	Vector of predicted values

calib
: A numeric vector of predicted values in the calibration partition, or a 2 column tibble or matrix with the first column being the predicted values and the second column being the truth values. If calib is a numeric vector, calib_truth must be provided.

calib_truth
: A numeric vector of true values in the calibration partition. Only required if calib is a numeric vector

alpha
: The confidence level for the prediction intervals. Must be a single numeric value between 0 and 1

ncs_type
: A string specifying the type of nonconformity score to use. Available options are:

  - `"absolute_error"`: $|y - \hat{y}|$
  - `"relative_error"`: $|y - \hat{y}|/\hat{y}$
  - `"zero_adjusted_relative_error"`: $|y - \hat{y}|/(\hat{y} + 1)$
  - `"heterogeneous_error"`: $|y - \hat{y}|/\sigma_{\hat{y}}$ absolute error divided by a measure of heteroskedasticity, computed as the predicted value from a linear model of the absolute error on the predicted values
  - `"raw_error"`: the signed error $y - \hat{y}$

  The default is `"absolute_error"`.

lower_bound
: Optional minimum value for the prediction intervals. If not provided, the minimum (true) value of the calibration partition will be used. Primarily useful when the possible outcome values are outside the range of values observed in the calibration set. If not provided, the minimum (true) value of the calibration partition will be used.

upper_bound
: Optional maximum value for the prediction intervals. If not provided, the maximum (true) value of the calibration partition will be used. Primarily useful when the possible outcome values are outside the range of values observed in the calibration set. If not provided, the maximum (true) value of the calibration partition will be used.

grid_size
: The number of points to use in the grid search between the lower and upper bound. Default is 10,000. A larger grid size increases the resolution of the prediction intervals but also increases computation time.

resolution
: Alternatively to grid_size. The minimum step size between grid points. Useful if the a specific resolution is desired. Default is NULL.

distance_weighted_cp
: Logical. If `TRUE`, weighted conformal prediction is performed where the nonconformity scores are weighted based on the distance between calibration and prediction points in feature space. Default is `FALSE`. See details for more information.

distance_features_calib
: A matrix, data frame, or numeric vector of features from which to compute distances when `distance_weighted_cp = TRUE`. This should contain the feature values for the calibration set. Must have the same number of rows as the calibration set. Can be the predicted values themselves, or any other features which give a meaningful distance measure.

`distance_features_pred`
: A matrix, data frame, or numeric vector of feature values for the prediction set. Must be the same features as specified in `distance_features_calib`. Required if `distance_weighted_cp = TRUE`.

`distance_type`
: The type of distance metric to use when computing distances between calibration and prediction points. Options are 'mahalanobis' (default) and 'euclidean'.

`normalize_distance`
: Either 'minmax', 'sd', or 'none'. Indicates if and how to normalize the distances when distance_weighted_cp is TRUE. Normalization helps ensure that distances are on a comparable scale across features. Default is 'none'.

`weight_function`
: A character string specifying the weighting kernel to use for distance-weighted conformal prediction. Options are:

  - `"gaussian_kernel"`: $w(d) = e^{-d^2}$
  - `"caucy_kernel"`: $w(d) = 1/(1 + d^2)$
  - `"logistic"`: $w(d) = 1//(1 + e^d)$
  - `"reciprocal_linear"`: $w(d) = 1/(1 + d)$

  The default is `"gaussian_kernel"`. Distances are computed as the Euclidean distance between the calibration and prediction feature vectors.

Details

This function computes prediction intervals using inductive conformal prediction. The calibration set must include predicted values and true values. These can be provided either as separate vectors (‘calib‘and ‘calib_truth‘) or as a two-column tibble or matrix where the first column contains the predicted values and the second column contains the true values. If ‘calib‘ is a numeric vector, ‘calib_truth‘ must also be provided.

Non-conformity scores are calculated using the specified ‘ncs_type‘, which determines how the prediction error is measured. Available options include:

- ‘"absolute_error"‘: the absolute difference between predicted and true values. - ‘"relative_error"‘: the absolute error divided by the true value. - ‘"za_relative_error"‘: zero-adjusted relative error, which replaces small or zero true values with a small constant to avoid division by zero. - ‘"heterogeneous_error"‘: absolute error scaled by a linear model of prediction error magnitude as a function of the predicted value. - ‘"raw_error"‘: the signed difference between predicted and true values.

These options provide flexibility to adapt to different patterns of prediction error across the outcome space.

To determine the prediction intervals, the function performs a grid search over a specified range of possible outcome values, identifying intervals that satisfy the desired confidence level of $1 - \alpha$. The user can define the range via the ‘lower_bound‘ and ‘upper_bound‘ parameters. If these are not supplied, the function defaults to using the minimum and maximum of the true values in the calibration data.

The resolution of the grid search can be controlled by either the ‘resolution‘ argument, which sets the minimum step size, or the ‘grid_size‘ argument, which sets the number of grid points. For wide prediction spaces, the grid search may be computationally intensive. In such cases, increasing the ‘resolution‘ or reducing the ‘grid_size‘ may improve performance.

When ‘distance_weighted_cp = TRUE‘, the function applies distance-weighted conformal prediction, which adjusts the influence of calibration non-conformity scores based on how similar each calibration point is to the target prediction. This approach preserves the distribution-free nature of conformal prediction while allowing intervals to adapt to local patterns, often yielding tighter and more responsive prediction sets in heterogeneous data environments.

Distances are computed between the feature matrices or vectors supplied via ‘distance_features_calib‘ and ‘distance_features_pred‘. These distances are then transformed into weights using the selected kernel in ‘weight_function‘, with rapidly decaying kernels (e.g., Gaussian) emphasizing strong locality and slower decays (e.g., reciprocal or Cauchy) providing smoother influence. Distances can be geographic coordinates, predicted values, or any other relevant features that capture similarity in the context of the prediction task. The distance metric is specified via ‘distance_type‘, with options for Mahalanobis or Euclidean distance. The default is Mahalanobis distance, which accounts for correlations between features. Normalization of distances can be applied using the ‘normalize_distance‘ parameter. Normalization is primarily useful for euclidean distances to ensure that features on different scales do not disproportionately influence the distance calculations.

Value

A tibble with the predicted values and the lower and upper bounds of the prediction intervals.

Examples

```
# Generate example data
library(dplyr)
library(tibble)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rlnorm(1000, meanlog = x1 + x2, sdlog = 0.5)
df <- tibble(x1, x2, y)
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model to the training data
mod <- lm(log(y) ~ x1 + x2, data=df_train)

# Generate predictions on the original y scale for the calibration data
calib_pred  <- exp(predict(mod, newdata=df_cal))
calib_truth <- df_cal$y

# Generate predictions for the test data
pred_test <- exp(predict(mod, newdata=df_test))

# Calculate prediction intervals using conformal prediction.
pinterval_conformal(pred_test,
calib = calib_pred,
calib_truth = calib_truth,
alpha = 0.1,
lower_bound = 0)
```

pinterval_mondrian *Mondrian Conformal Prediction Intervals for Continuous Predictions*

Description

This function calculates Mondrian conformal prediction intervals with a confidence level of 1-alpha for a vector of (continuous) predicted values using inductive conformal prediction on a Mondrian class-by-class basis. The intervals are computed using a calibration set with predicted and true values and their associated classes. The function returns a tibble containing the predicted values along with the lower and upper bounds of the prediction intervals. Mondrian conformal prediction intervals are useful when the prediction error is not constant across groups or classes, as they allow for locally valid coverage by ensuring that the coverage level $1 - \alpha$ holds within each class—assuming exchangeability of non-conformity scores within classes.

Usage

```
pinterval_mondrian(
  pred,
  pred_class = NULL,
  calib = NULL,
  calib_truth = NULL,
  calib_class = NULL,
  alpha = 0.1,
  ncs_type = c("absolute_error", "relative_error", "za_relative_error",
    "heterogeneous_error", "raw_error"),
  lower_bound = NULL,
  upper_bound = NULL,
  grid_size = 10000,
  resolution = NULL,
  distance_weighted_cp = FALSE,
  distance_features_calib = NULL,
  distance_features_pred = NULL,
  distance_type = c("mahalanobis", "euclidean"),
  normalize_distance = "none",
 weight_function = c("gaussian_kernel", "caucy_kernel", "logistic", "reciprocal_linear")
)
```

Arguments

pred	Vector of predicted values
pred_class	A vector of class identifiers for the predicted values. This is used to group the predictions by class for Mondrian conformal prediction.
calib	A numeric vector of predicted values in the calibration partition, or a 2 column tibble or matrix with the first column being the predicted values and the second column being the truth values. If calib is a numeric vector, calib_truth must be provided.

`calib_truth` A numeric vector of true values in the calibration partition. Only required if calib is a numeric vector

`calib_class` A vector of class identifiers for the calibration set.

`alpha` The confidence level for the prediction intervals. Must be a single numeric value between 0 and 1

`ncs_type` A string specifying the type of nonconformity score to use. Available options are:

- `"absolute_error"`: $|y - \hat{y}|$
- `"relative_error"`: $|y - \hat{y}|/\hat{y}$
- `"zero_adjusted_relative_error"`: $|y - \hat{y}|/(\hat{y} + 1)$
- `"heterogeneous_error"`: $|y - \hat{y}|/\sigma_{\hat{y}}$ absolute error divided by a measure of heteroskedasticity, computed as the predicted value from a linear model of the absolute error on the predicted values
- `"raw_error"`: the signed error $y - \hat{y}$

The default is `"absolute_error"`.

`lower_bound` Optional minimum value for the prediction intervals. If not provided, the minimum (true) value of the calibration partition will be used. Primarily useful when the possible outcome values are outside the range of values observed in the calibration set. If not provided, the minimum (true) value of the calibration partition will be used.

`upper_bound` Optional maximum value for the prediction intervals. If not provided, the maximum (true) value of the calibration partition will be used. Primarily useful when the possible outcome values are outside the range of values observed in the calibration set. If not provided, the maximum (true) value of the calibration partition will be used.

`grid_size` The number of points to use in the grid search between the lower and upper bound. Default is 10,000. A larger grid size increases the resolution of the prediction intervals but also increases computation time.

`resolution` Alternatively to grid_size. The minimum step size between grid points. Useful if the a specific resolution is desired. Default is NULL.

`distance_weighted_cp`
Logical. If `TRUE`, weighted conformal prediction is performed where the nonconformity scores are weighted based on the distance between calibration and prediction points in feature space. Default is `FALSE`. See details for more information.

`distance_features_calib`
A matrix, data frame, or numeric vector of features from which to compute distances when `distance_weighted_cp = TRUE`. This should contain the feature values for the calibration set. Must have the same number of rows as the calibration set. Can be the predicted values themselves, or any other features which give a meaningful distance measure.

`distance_features_pred`
A matrix, data frame, or numeric vector of feature values for the prediction set. Must be the same features as specified in `distance_features_calib`. Required if `distance_weighted_cp = TRUE`.

`distance_type` The type of distance metric to use when computing distances between calibration and prediction points. Options are 'mahalanobis' (default) and 'euclidean'.

`normalize_distance`
Either 'minmax', 'sd', or 'none'. Indicates if and how to normalize the distances when distance_weighted_cp is TRUE. Normalization helps ensure that distances are on a comparable scale across features. Default is 'none'.

`weight_function`
A character string specifying the weighting kernel to use for distance-weighted conformal prediction. Options are:

- `"gaussian_kernel"`: $w(d) = e^{-d^2}$
- `"caucy_kernel"`: $w(d) = 1/(1 + d^2)$
- `"logistic"`: $w(d) = 1//(1 + e^d)$
- `"reciprocal_linear"`: $w(d) = 1/(1 + d)$

The default is `"gaussian_kernel"`. Distances are computed as the Euclidean distance between the calibration and prediction feature vectors.

Details

'pinterval_mondrian()' extends [pinterval_conformal()] to the Mondrian setting, where prediction intervals are calibrated separately within user-defined groups (often called "Mondrian categories"). Instead of pooling all calibration residuals into a single reference distribution, the method constructs a separate non-conformity distribution for each subgroup defined by a grouping variable (e.g., region, regime type, or income category). This allows the intervals to adapt to systematic differences in error magnitude or variance across groups and targets coverage conditional on group membership. It is especially useful when prediction error varies systematically across known categories, allowing for class-conditional validity by ensuring that the prediction intervals attain the desired coverage level $1 - \alpha$ within each class—under the assumption of exchangeability within classes.

Conceptually, the underlying inductive conformal machinery is the same as in [pinterval_conformal()], but applied within groups rather than globally. For a detailed description of non-conformity scores, distance-weighting, and the general conformal prediction framework, see [pinterval_conformal()].

For 'pinterval_mondrian()', the calibration set must include predicted values, true values, and corresponding class labels. These can be supplied as separate vectors ('calib', 'calib_truth', and 'calib_class') or as a single three-column matrix or tibble.

Value

A tibble with predicted values, lower and upper prediction interval bounds, and class labels.

See Also

`pinterval_conformal`

Examples

```
# Generate synthetic data
library(dplyr)
library(tibble)
```

```
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
group <- sample(c("A", "B", "C"), size = 1000, replace = TRUE)
mu <- ifelse(group == "A", 1 + x1 + x2,
      ifelse(group == "B", 2 + x1 + x2,
                           3 + x1 + x2))
y <- rlnorm(1000, meanlog = mu, sdlog = 0.4)

df <- tibble(x1, x2, group, y)
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model to the training data
mod <- lm(log(y) ~ x1 + x2, data = df_train)

# Generate predictions
calib <- exp(predict(mod, newdata = df_cal))
calib_truth <- df_cal$y
calib_class <- df_cal$group

pred_test <- exp(predict(mod, newdata = df_test))
pred_test_class <- df_test$group

# Apply Mondrian conformal prediction
pinterval_mondrian(pred = pred_test,
pred_class = pred_test_class,
calib = calib,
calib_truth = calib_truth,
calib_class = calib_class,
alpha = 0.1)
```

`pinterval_parametric` *Parametric Prediction Intervals for Continuous Predictions*

Description

This function computes parametric prediction intervals at a confidence level of $1 - \alpha$ for a vector of continuous predictions. The intervals are based on a user-specified probability distribution and associated parameters, either estimated from calibration data or supplied directly. Supported distributions include common options like the normal, log-normal, gamma, beta, and negative binomial, as well as any user-defined distribution with a quantile function. Prediction intervals are calculated by evaluating the appropriate quantiles for each predicted value.

Usage

```
pinterval_parametric(
  pred,
```

```
  calib = NULL,
  calib_truth = NULL,
 dist = c("norm", "lnorm", "exp", "pois", "nbinom", "gamma", "chisq", "logis", "beta"),
  pars = list(),
  alpha = 0.1
)
```

Arguments

pred	Vector of predicted values
calib	A numeric vector of predicted values in the calibration partition, or a 2 column tibble or matrix with the first column being the predicted values and the second column being the truth values. If calib is a numeric vector, calib_truth must be provided.
calib_truth	A numeric vector of true values in the calibration partition. Only required if calib is a numeric vector
dist	Distribution to use for the prediction intervals. Can be a character string matching any available distribution in R or a function representing a distribution, e.g. ‘qnorm‘, ‘qgamma‘, or a user defined quantile function. Default options are ’norm’, ’lnorm’,’exp, ’pois’, ’nbinom’, ’chisq’, ’gamma’, ’logis’, and ’beta’ for which parameters can be computed from the calibration set. If a custom function is provided, parameters need to be provided in ‘pars‘.
pars	List of named parameters for the distribution for each prediction. Not needed if calib is provided and the distribution is one of the default options. If a custom distribution function is provided, this list should contain the parameters needed for the quantile function, with names matching the corresponding arguments for the parameter names of the distribution function. See details for more information.
alpha	The confidence level for the prediction intervals. Must be a single numeric value between 0 and 1

Details

This function supports a wide range of distributions for constructing prediction intervals. Built-in support is provided for the following distributions: ‘"norm"‘, ‘"lnorm"‘, ‘"exp"‘, ‘"pois"‘, ‘"nbinom"‘, ‘"chisq"‘, ‘"gamma"‘, ‘"logis"‘, and ‘"beta"‘. For each of these, parameters can be automatically estimated from a calibration set if not supplied directly via the ‘pars‘ argument.

The calibration set (‘calib‘ and ‘calib_truth‘) is used to estimate error dispersion or shape parameters. For example: - **Normal**: standard deviation of errors - **Log-normal**: standard deviation of log-errors - **Gamma**: dispersion via ‘glm‘ - **Negative binomial**: dispersion via ‘glm.nb()‘ - **Beta**: precision estimated from error variance

If ‘pars‘ is supplied, it should be a list of named arguments corresponding to the distribution’s quantile function. Parameters may be scalars or vectors (one per prediction). When both ‘pars‘ and ‘calib‘ are provided, the values in ‘pars‘ are used.

Users may also specify a custom distribution by passing a quantile function directly (e.g., a function with the signature ‘function(p, ...)‘) as the ‘dist‘ argument, in which case ‘pars‘ must be provided explicitly.

Value

A tibble with the predicted values and the lower and upper bounds of the prediction intervals

Examples

```
library(dplyr)
library(tibble)

# Simulate example data
set.seed(123)
x1 <- runif(1000)
x2 <- runif(1000)
y <- rlnorm(1000, meanlog = x1 + x2, sdlog = 0.5)
df <- tibble(x1, x2, y)

# Split into training, calibration, and test sets
df_train <- df %>% slice(1:500)
df_cal <- df %>% slice(501:750)
df_test <- df %>% slice(751:1000)

# Fit a model on the log-scale
mod <- lm(log(y) ~ x1 + x2, data = df_train)

# Generate predictions
pred_cal <- exp(predict(mod, newdata = df_cal))
pred_test <- exp(predict(mod, newdata = df_test))

# Estimate log-normal prediction intervals from calibration data
log_resid_sd <- sqrt(mean((log(pred_cal) - log(df_cal$y))^2))
pinterval_parametric(
  pred = pred_test,
  dist = "lnorm",
  pars = list(meanlog = log(pred_test), sdlog = log_resid_sd)
)

# Alternatively, use calibration data directly to estimate parameters
pinterval_parametric(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  dist = "lnorm"
)

# Use the normal distribution with direct parameter input
norm_sd <- sqrt(mean((pred_cal - df_cal$y)^2))
pinterval_parametric(
  pred = pred_test,
  dist = "norm",
  pars = list(mean = pred_test, sd = norm_sd)
)

# Use the gamma distribution with parameters estimated from calibration data
```

```
pinterval_parametric(
  pred = pred_test,
  calib = pred_cal,
  calib_truth = df_cal$y,
  dist = "gamma"
)
```

Index

∗ **datasets**
county_turnout, 2
county_turnout_panel, 4
elections, 8

county_turnout, 2, *4*, *7*
county_turnout_panel, *3*, 4

elections, 8

interval_coverage, 9
interval_miscoverage, 10
interval_score, 12
interval_width, 14

pinterval_bccp, 15
pinterval_bootstrap, 19
pinterval_ccp, 22
pinterval_conformal, *18*, *26*, 27, *33*
pinterval_mondrian, *26*, 31
pinterval_parametric, 34