Package: wisclabmisc (via r-universe)

September 10, 2024

Type Package

Title Tools to Support the 'WiscLab'

Version 0.1.0

Description A collection of 'R' functions for use (and re-use) across 'WiscLab' projects. These are analysis or presentation oriented functions--that is, they are not for data reading or data cleaning.

License MIT + file LICENSE

URL https://github.com/tjmahr/wisclabmisc, https://www.tjmahr.com/wisclabmisc/

BugReports https://github.com/tjmahr/wisclabmisc/issues

Depends R (>= 4.1)

Imports dplyr, gamlss (>= 5.2.0), magrittr, ordinal, pROC, purrr, rlang (>= 0.4.11), rsample, sessioninfo, stringr, tibble, tidyr, tidyselect, usethis

Suggests gamlss.dist, ggplot2, knitr, logitnorm, nlme, numDeriv, rmarkdown, roxygen2, rstanarm, testthat, tidyverse

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Repository https://tjmahr.r-universe.dev

RemoteUrl https://github.com/tjmahr/wisclabmisc

RemoteRef HEAD

RemoteSha 2d2df5af2ec937c763fdd4a370514b7678bd08c7

Contents

`check_sample_centiles` *Compute the percentage of points under each centile line*

Description

Compute the percentage of points under each centile line

Usage

```
check_sample_centiles(
  data,
  model,
  var_x,
  var_y,
  centiles = c(5, 10, 25, 50, 75, 90, 95)
)
```

Arguments

`data`	a dataset used to fit a model. If the dataframe is grouped with `dplyr::group_by()`, sample centiles are computed for each group.
`model`	a gamlss model prepared by `mem_gamlss()`
`var_x`, `var_y`	bare column names of the predictor and outcome variables
`centiles`	centiles to use for prediction. Defaults to `c(5, 10, 25, 50, 75, 90, 95)`.

Value

a tibble the number of points and the percentage of points less than or equal to each quantile value.

`chrono_age` *Compute chronological age in months*

Description

Ages are rounded down to the nearest month. A difference of 20 months, 29 days is interpreted as 20 months.

Usage

```
chrono_age(t1, t2)
```

Arguments

`t1`, `t2`	dates in "yyyy-mm-dd" format

Value

the chronological ages in months. NA is returned if the age cannot be computed.

Examples

```
# Two years exactly
chrono_age("2014-01-20", "2012-01-20")
#> 24

# Shift a year
chrono_age("2014-01-20", "2013-01-20")
#> 12
chrono_age("2014-01-20", "2011-01-20")
#> 36

# Shift a month
chrono_age("2014-01-20", "2012-02-20")
#> 23
chrono_age("2014-01-20", "2011-12-20")
```

```
#> 25

# 3 months exactly
chrono_age("2014-05-10", "2014-02-10")
#> 3

# Borrow a month when the earlier date has a later day
chrono_age("2014-05-10", "2014-02-11")
#> 2, equal to 2 months, 29 days rounded down to nearest month

# Inverted argument order
chrono_age("2012-01-20", "2014-01-20")
#> 24

# Multiple dates
t1 <- c("2012-01-20", "2014-02-10", "2010-10-10")
t2 <- c("2014-01-20", "2014-05-10", "2014-11-10")
chrono_age(t1, t2)
#> [1] 24  3 49
```

`compute_empirical_roc`

Create an ROC curve from observed data

Description

Create an ROC curve from observed data

Usage

```
compute_empirical_roc(
  data,
  response,
  predictor,
  direction = "auto",
  best_weights = c(1, 0.5),
  ...
)
```

Arguments

`data`	a dataframe containing responses (groupings) and predictor variable
`response`	a bare column name with the group status (control vs. cases)
`predictor`	a bare column name with the predictor to use for classification
`direction`	`direction` to set for the for `pROC::roc()`. Defaults to `"auto"`.
`best_weights`	weights for computing the best ROC curve points. Defaults to `c(1, .5)`, which are the defaults used by `pROC::coords()`.
`...`	additional arguments passed to `pROC::roc()`.

Value

a new dataframe of ROC coordinates is returned with columns for the predictor variable, `.sensitivities`, `.specificities`, `.auc`, `.direction`, `.controls`, `.cases`, `.n_controls`, `.n_cases`, `.is_best_youden` and `.is_best_closest_topleft`.

Examples

```
set.seed(100)
x1 <- rnorm(100, 4, 1)
x2 <- rnorm(100, 2, .5)
both <- c(x1, x2)
steps <- seq(min(both), max(both), length.out = 200)
d1 <- dnorm(steps, mean(x1), sd(x1))
d2 <- dnorm(steps, mean(x2), sd(x2))
data <- tibble::tibble(
  y = steps,
  d1 = d1,
  d2 = d2,
  outcome = rbinom(200, 1, prob = 1 - (d1 / (d1 + d2))),
  group = ifelse(outcome, "case", "control")
)

# get an ROC on the fake data
compute_empirical_roc(data, outcome, y)
# this guess the cases and controls from the group name and gets it wrong
compute_empirical_roc(data, group, y)
# better
compute_empirical_roc(data, group, y, levels = c("control", "case"))
```

`compute_predictive_value_from_rates` *Compute positive and negative predictive value*

Description

Compute positive and negative predictive value

Usage

```
compute_predictive_value_from_rates(sensitivity, specificity, prevalence)
```

Arguments

`sensitivity`, `specificity`, `prevalence` vectors of confusion matrix rates

Details

These vectors passed into this function should be some common length and/or length 1. For example, 4 sensitivities, 4 specificities and 1 incidence will work because the sensitivities and specificities have a common length and we can safely recycle (reuse) the incidence value. But 4 sensitivities, 2 specificities, and 1 incidence will *fail* because there is not a common length.

Value

a tibble with the columns `sensitivity`, `specificity`, `prevalence`, `ppv`, `npv` where `ppv` and `npv` are the positive predictive value and the negative predictive value.

Examples

```
compute_predictive_value_from_rates(
  sensitivity = .9,
  specificity = .8,
  prevalence = .05
)

compute_predictive_value_from_rates(
  sensitivity = .67,
  specificity = .53,
  prevalence = c(.15, .3)
)
```

---

compute_smooth_density_roc

Create an ROC curve from smoothed densities

Description

Create an ROC curve from smoothed densities

Usage

```
compute_smooth_density_roc(
  data,
  controls,
  cases,
  along = NULL,
  best_weights = c(1, 0.5),
  direction = "auto",
  ...
)
```

Arguments

`data`	a dataframe containing densities
`controls`, `cases`	bare column name for the densities of the control group
`along`	optional bare column name for the response values
`best_weights`	weights for computing the best ROC curve points. Defaults to `c(1, .5)`, which are the defaults used by `pROC::coords()`.
`direction`	`direction` to set for the for `pROC::roc()`. Defaults to `"auto"`.
`...`	additional arguments. Not used currently.

Value

the dataframe is updated with new columns for the `.sensitivities`, `.specificities`, `.auc`, `.roc_row`, `.is_best_youden` and `.is_best_closest_topleft`.

Examples

```
set.seed(100)
x1 <- rnorm(100, 4, 1)
x2 <- rnorm(100, 2, .5)
both <- c(x1, x2)
steps <- seq(min(both), max(both), length.out = 200)
d1 <- dnorm(steps, mean(x1), sd(x1))
d2 <- dnorm(steps, mean(x2), sd(x2))
data <- tibble::tibble(
  y = steps,
  d1 = d1,
  d2 = d2,
  outcome = rbinom(200, 1, prob = 1 - (d1 / (d1 + d2))),
  group = ifelse(outcome, "case", "control")
)
compute_smooth_density_roc(data, d1, d2)
compute_smooth_density_roc(data, d1, d2, along = y)

# terrible ROC because the response is not present (just the densities)
data_shuffled <- data[sample(seq_len(nrow(data))), ]
compute_smooth_density_roc(data_shuffled, d1, d2)

# sorted along response first: correct AUC
compute_smooth_density_roc(data_shuffled, d1, d2, along = y)
```

`data_example_intelligibility_by_length` *Simulated intelligibility scores by utterance length*

Description

A dataset of simulated intelligibility scores for testing and demonstrating modeling functions. These were created by fitting a Bayesian model of the raw Hustad and colleagues (2020) and drawing 1 sample from the posterior distribution of expected predictions (i.e., "epreds). In other words, these values are model predictions of the original dataset. They are correlated with original dataset values at $r = .86$. We might think of the simulation as adding random noise to the original dataset.

Usage

```
data_example_intelligibility_by_length
```

Format

A data frame with 694 rows and 5 variables:

child identifier for the child

age_months child's age in months

length_longest length of the child's longest utterance

tocs_level utterance length

sim_intelligibility child's intelligibility for the given utterance length (proportion of words said by the child that were correctly transcribed by two listeners)

References

Hustad, K. C., Mahr, T., Natzke, P. E. M., & Rathouz, P. J. (2020). Development of Speech Intelligibility Between 30 and 47 Months in Typically Developing Children: A Cross-Sectional Study of Growth. *Journal of Speech, Language, and Hearing Research*, *63*(6), 1675–1687. https://doi.org/10.1044/2020_JSLHR-20-00008

`data_fake_intelligibility`

Fake intelligibility data

Description

A dataset of fake intelligibility scores for testing and demonstrating modeling functions. These were created by randomly sampling 200 rows of an intelligibility dataset and adding random noise to the `age_months` and `intelligibility` variables. These values do not measure any real children but represent plausible age and intelligibility measurements from our kind of work.

Usage

```
data_fake_intelligibility
```

Format

A data frame with 200 rows and 2 variables:

age_months child's age in months

intelligibility child's intelligibility (proportion of words said by the child that were correctly transcribed by two listeners)

---

data_fake_rates *Fake speaking rate data*

Description

A dataset of fake speaking rate measures for testing and demonstrating modeling functions. These were created by randomly sampling 200 rows of a speaking rate dataset and adding random noise to the `age_months` and `speaking_sps` variables. These values do not measure any real children but represent plausible age and rate measurements from our kind of work.

Usage

```
data_fake_rates
```

Format

A data frame with 200 rows and 2 variables:

age_months child's age in months

speaking_sps child's speaking rate in syllables per second

---

data_features_consonants *Phonetic features of consonants and vowels*

Description

These are two dataframes that contain conventional phonetic features of the consonants and vowels used by CMU phonetic alphabet.

Usage

```
data_features_consonants

data_features_vowels
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 24 rows and 10 columns.

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 17 rows and 12 columns.

Details

Most of the features are self-evident and definitional. For example, /p/ is *the* bilabial voiceless stop. For fuzzier features, I consulted the general IPA chart and the Wikipedia page on English phonology. These issues included things like: *what are the lax vowels again?* or *the last two rows of the consonant tables are approximants, so /r,l,j/ are approximants.*

Some features have alternative feature sets in order to accommodate degrees of aggregation. For example, /r,l,j,w/ are *approximant* in `manner` but divided into *liquid* and *glide* in `manner_alt`.

Consonants:

`data_features_consonants` is a dataframe with 24 rows and 10 variables.

```
knitr::kable(data_features_consonants)
```

phone	cmubet	wiscbet	voicing	manner	manner_alt	place	place_fct	sonorance	sono
p	P	p	voiceless	stop	stop	labial	labial	obstruent	obst
b	B	b	voiced	stop	stop	labial	labial	obstruent	obst
t	T	t	voiceless	stop	stop	alveolar	alveolar	obstruent	obst
d	D	d	voiced	stop	stop	alveolar	alveolar	obstruent	obst
k	K	k	voiceless	stop	stop	velar	velar	obstruent	obst
g	G	g	voiced	stop	stop	velar	velar	obstruent	obst
t	CH	tsh	voiceless	affricate	affricate	postalveolar	postalveolar	obstruent	strid
d	JH	dzh	voiced	affricate	affricate	postalveolar	postalveolar	obstruent	strid
m	M	m	voiced	nasal	nasal	labial	labial	sonorant	sono
n	N	n	voiced	nasal	nasal	alveolar	alveolar	sonorant	sono
ŋ	NG	ng	voiced	nasal	nasal	velar	velar	sonorant	sono
f	F	f	voiceless	fricative	fricative	labiodental	labiodental	obstruent	strid
v	V	v	voiced	fricative	fricative	labiodental	labiodental	obstruent	strid
θ	TH	th	voiceless	fricative	fricative	dental	dental	obstruent	obst
ð	DH	dh	voiced	fricative	fricative	dental	dental	obstruent	obst
s	S	s	voiceless	fricative	fricative	alveolar	alveolar	obstruent	strid
z	Z	z	voiced	fricative	fricative	alveolar	alveolar	obstruent	strid
ʃ	SH	sh	voiceless	fricative	fricative	postalveolar	postalveolar	obstruent	strid
ʒ	ZH	zh	voiced	fricative	fricative	postalveolar	postalveolar	obstruent	strid
h	HH	h	voiceless	fricative	fricative	glottal	glottal	obstruent	obst
l	L	l	voiced	approximant	liquid	alveolar	alveolar	sonorant	sono
r	R	r	voiced	approximant	liquid	postalveolar	postalveolar	sonorant	sono
w	W	w	voiced	approximant	glide	labiovelar	NA	sonorant	sono
j	Y	j	voiced	approximant	glide	palatal	palatal	sonorant	sono

Description of each column:

phone phone in IPA
cmubet phone in the CMU alphabet
wiscbet phone in an older system used by our lab
voicing *voiced* versus *voiceless*
voicing_alt *spread_glottis* versus *plain*

manner manner of articulation

manner_alt alternative manner coding that separates *approximants* into *liquids* and *glides*

place place of articulation

place_fct place coded as a factor and ordered based on frontness of the articulators. *labiovelar* is recoded as `NA`.

sonorance *obstruent* versus *sonorant*

sonorance_alt *obstruant* versus *sonorant* versus *strident*.

Levels of the factor columns:

```
data_features_consonants |>
  lapply(levels) |>
  Filter(length, x = _)
#> $place_fct
#> [1] "labial"       "labiodental"  "dental"       "alveolar"     "postalveolar"
#> [6] "palatal"      "velar"        "glottal"
```

Considerations about consonant features:

The CMU alphabet does not include a glottal stop.

Here /f,v/ are coded as *strident* following Wikipedia and *Sound Pattern of English*. If this feature value doesn't seem right, we should probably use an alternative feature of *sibilant* for the stridents minus /f,v/.

The alternative voicing scheme was suggested by a colleague because of how the voice-voiceless phonetic contrast is achieved with different articulatory strategies in different languages. Note that `voicing_alt` does not assign a feature to nasals or approximants.

Vowels:

`data_features_vowels` is a dataframe with 17 rows and 11 variables.

```
knitr::kable(data_features_vowels)
```

phone	cmubet	wiscbet	hint	manner	manner_alt	tenseness	height	height_fct	backness	b
i	IY	i	beat	vowel	vowel	tense	high	high	front	f
	IH	I	bit	vowel	vowel	lax	high	high	front	f
e	EY	eI	bait	vowel	vowel	tense	mid	mid	front	f
	EH	E	bet	vowel	vowel	lax	mid	mid	front	f
æ	AE	ae	bat	vowel	vowel	lax	low	low	front	f
	AH	^	but	vowel	vowel	lax	mid	mid	central	c
ə	AH	4	comma	vowel	vowel	lax	mid	mid	central	c
u	UW	u	boot	vowel	vowel	tense	high	high	back	b
	UH	U	book	vowel	vowel	lax	high	high	back	b
o	OW	oU	boat	vowel	vowel	tense	mid	mid	back	b
	AO	c	bought	vowel	vowel	tense	mid	mid	back	b
	AA	@	bot	vowel	vowel	tense	low	low	back	b
a	AW	@U	bout	vowel	diphthong	diphthong	diphthong	NA	diphthong	N
a	AY	@I	bite	vowel	diphthong	diphthong	diphthong	NA	diphthong	N
	OY	cI	boy	vowel	diphthong	diphthong	diphthong	NA	diphthong	N

ER	3^	letter	vowel	r-colored	r-colored	mid	mid	central	c
ER	4^	burt	vowel	r-colored	r-colored	mid	mid	central	c

Description of each column:

phone phone in IPA

cmubet phone in the CMU alphabet

wiscbet phone in an older system used by our lab

hint a word containing the selected vowel

manner manner of articulation

manner_alt alternative manner with *vowel*, *diphthong* and *r-colored*

tenseness *tense* versus *lax* (versus *diphthong* and *r-colored*)

height vowel height

height_fct height coded as a factor ordered *high*, *mid*, *low*. *diphthong* is recoded to `NA`.

backness vowel backness

backness_fct backness coded as a factor ordered *front*, *central*, *back*. *diphthong* is recoded to `NA`.

rounding *unrounded* versus *rounded* (versus *diphthong* and *r-colored*)

Levels of the factor columns:

```
data_features_vowels |>
  lapply(levels) |>
  Filter(length, x = _)
#> $height_fct
#> [1] "high" "mid"  "low"
#>
#> $backness_fct
#> [1] "front"   "central" "back"
```

Considerations about vowel features:

I don't consider /e / and /o / to be diphthongs, but perhaps `manner_alt` could encode the difference of these vowels from the others.

In the CMU alphabet and ARPAbet, vowels can include a number to indicate vowel stress, so `AH1` or `AH2` is / / but `AH0` is /ə/.

The vowel features for General American English, according to Wikipedia, are as follows:

I adapted these features in this way:

- *tense* and *lax* features were directly borrowed. Diphthongs and r-colored vowels are were not assign a tenseness.
- / , / raised to *mid* (following the general IPA chart)
- / / moved to *back* (following the general IPA)
- diphthongs have no backness or height
- r-colored vowels were given the backness and height of the / ,ə/

Based on the assumption that / ,ə/ are the same general vowel with differing stress, these vowels have the same features. This definition clashes with the general IPA chart which

places / / as a back vowel. However, / / is a conventional notation. Quoting Wikipedia again: "Although the notation / / is used for the vowel of STRUT in RP and General American, the actual pronunciation is closer to a near-open central vowel [ ] in RP and advanced back [ ] in General American." That is, / / is fronted in American English (hence, *mid*) in American English.

`fit_beta_gamlss` *Fit a beta regression model (for intelligibility)*

Description

The function fits the same type of GAMLSS model as used in Hustad and colleagues (2021): A beta regression model (via `gamlss.dist::BE()`) with natural cubic splines on the mean (mu) and scale (sigma). This model is fitted using this package's `mem_gamlss()` wrapper function.

Usage

```
fit_beta_gamlss(data, var_x, var_y, df_mu = 3, df_sigma = 2, control = NULL)

fit_beta_gamlss_se(
  data,
  name_x,
  name_y,
  df_mu = 3,
  df_sigma = 2,
  control = NULL
)

predict_beta_gamlss(newdata, model, centiles = c(5, 10, 50, 90, 95))

optimize_beta_gamlss_slope(
  model,
  centiles = 50,
  interval = c(30, 119),
  maximum = TRUE
)

uniroot_beta_gamlss(model, centiles = 50, targets = 0.5, interval = c(30, 119))
```

Arguments

`data`	a data frame
`var_x`, `var_y`	(unquoted) variable names giving the predictor variable (e.g., `age`) and outcome variable (.e.g, `intelligibility`).

`df_mu`, `df_sigma` degrees of freedom. If `0` is used, the `splines::ns()` term is dropped from the model formula for the parameter.

`control` a `gamlss::gamlss.control()` controller. Defaults to `NULL` which uses default settings, except for setting trace to `FALSE` to silence the output from gamlss.

`name_x`, `name_y` quoted variable names giving the predictor variable (e.g., `"age"`) and outcome variable (.e.g, `"intelligibility"`). These arguments apply to `fit_beta_gamlss_se()`.

`newdata` a one-column dataframe for predictions

`model` a model fitted by `fit_beta_gamlss()`

`centiles` centiles to use for prediction. Defaults to `c(5, 10, 50, 90, 95)` for `predict_beta_gamlss()`. Defaults to `50` for `optimize_beta_gamlss_slope()` and `uniroot_beta_gamlss()`, although both of these functions support multiple centile values.

`interval` for `optimize_beta_gamlss_slope()`, the range of `x` values to optimize over. For `uniroot_beta_gamlss()`, the range of `x` values to search for roots (target y values) in.

`maximum` for `optimize_beta_gamlss_slope()`, whether to find the maximum slope (`TRUE`) or minimum slope (`FALSE`).

`targets` for `uniroot_beta_gamlss()`, the target y values to use as roots. By default, .5 is used, so that `uniroot_beta_gamlss()` returns the `x` value where the y value is .5. Multiple targets are supported.

Details

There are two versions of this function. The main version is `fit_beta_gamlss()`, and it works with unquoted column names (e.g., `age`). The alternative version is `fit_beta_gamlss_se()`; the final "se" stands for "Standard Evaluation". This designation means that the variable names must be given as strings (so, the quoted `"age"` instead of bare name `age`). This alternative version is necessary when we fit several models using parallel computing with `furrr::future_map()` (as when using bootstrap resampling).

`predict_centiles()` will work with this function, but it will likely throw a warning message. Therefore, `predict_beta_gamlss()` provides an alternative way to compute centiles from the model. This function manually computes the centiles instead of relying on `gamlss::centiles()`. The main difference is that new x values go through `splines::predict.ns()` and then these are multiplied by model coefficients.

`optimize_beta_gamlss_slope()` computes the point (i.e., age) and rate of steepest growth for different quantiles. This function wraps over the following process:

- an internal prediction function computes a quantile at some `x` from model coefficients and spline bases.
- another internal function uses `numDeriv::grad()` to get the gradient of this prediction function for `x`.
- `optimize_beta_gamlss_slope()` uses `stats::optimize()` on the gradient function to find the `x` with the maximum or minimum slope.

`uniroot_beta_gamlss()` also uses this internal prediction function to find when a quantile growth curve crosses a given value. `stats::uniroot()` finds where a function crosses 0 (a root). If we modify our prediction function to always subtract .5 at the end, then the root for this prediction function would be the x value where the predicted value crosses .5. That's the idea behind how `uniroot_beta_gamlss()` works. In our work, we would use this approach to find, say, the age (root) when children in the 10th percentile (`centiles`) cross 50% intelligibility (`targets`).

GAMLSS does beta regression differently:

This part is a brief note that GAMLSS uses a different parameterization of the beta distribution for its beta family than other packages.

The canonical parameterization of the beta distribution uses shape parameters α and β and the probability density function:

$$f(y;\alpha,\beta) = \frac{1}{B(\alpha,\beta)} y^{\alpha-1}(1-y)^{\beta-1}$$

where B is the beta function.

For beta regression, the distribution is reparameterized so that there is a mean probability μ and some other parameter that represents the spread around that mean. In GAMLSS (`gamlss.dist::BE()`), they use a scale parameter σ (larger values mean more spread around mean). Everywhere else—`betareg::betareg()` and `rstanarm::stan_betareg()` in `vignette("betareg", "betareg")`, `brms::Beta()` in `vignette("brms_families", "brms")`, `mgcv::betar()`—it's a precision parameter ϕ (larger values mean more precision, less spread around mean). Here is a comparison:

$$\text{betareg, brms, mgcv, etc.} \mu = \alpha/(\alpha+\beta) \phi = \alpha + b \mathsf{E}(y) = \mu \mathsf{VAR}(y) = \mu(1-\mu)/(1+\phi)$$

$$\text{GAMLSS} \mu = \alpha/(\alpha+\beta) \sigma = (1/(\alpha+\beta+1)) \cdot 5 \mathsf{E}(y) = \mu \mathsf{VAR}(y) = \mu(1-\mu)\sigma^2$$

Value

for `fit_beta_gamlss()` and `fit_beta_gamlss_se()`, a `mem_gamlss()`-fitted model. The `.user` data in the model includes degrees of freedom for each parameter and the `splines::ns()` basis for each parameter. For `predict_beta_gamlss()`, a dataframe containing the model predictions for mu and sigma, plus columns for each centile in `centiles`. For `optimize_beta_gamlss_slope()`, a dataframe with the optimized `x` values (`maximum` or `minimum`), the gradient at that `x` value (`objective`), and the quantile (`quantile`). For `uniroot_beta_gamlss()`, a dataframe one row per quantile/target combination with the results of calling `stats::uniroot()`. The `root` column is the `x` value where the `quantile` curve crosses the `target` value.

Source

Associated article: https://doi.org/10.1044/2021_JSLHR-21-00142

Examples

```
data_fake_intelligibility

m <- fit_beta_gamlss(
  data_fake_intelligibility,
  age_months,
  intelligibility
)

# using "qr" in summary() just to suppress a warning message
summary(m, type = "qr")

# Alternative interface
d <- data_fake_intelligibility
m2 <- fit_beta_gamlss_se(
  data = d,
  name_x = "age_months",
  name_y = "intelligibility"
)
coef(m2) == coef(m)

# how to use control to change gamlss() behavior
m_traced <- fit_beta_gamlss(
  data_fake_intelligibility,
  age_months,
  intelligibility,
  control = gamlss::gamlss.control(n.cyc = 15, trace = TRUE)
)

# The `.user` space includes the spline bases, so that we can make accurate
# predictions of new xs.
names(m$.user)

# predict logit(mean) at 55 months:
logit_mean_55 <- cbind(1, predict(m$.user$basis_mu, 55)) %*% coef(m)
logit_mean_55
stats::plogis(logit_mean_55)

# But predict_gen_gamma_gamlss() does this work for us and also provides
# centiles
new_ages <- data.frame(age_months = 48:71)
centiles <- predict_beta_gamlss(new_ages, m)
centiles

# Confirm that the manual prediction matches the automatic one
centiles[centiles$age_months == 55, "mu"]
stats::plogis(logit_mean_55)

if(requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot(pivot_centiles_longer(centiles)) +
    aes(x = age_months, y = .value) +
```

```
    geom_line(aes(group = .centile, color = .centile_pair)) +
    geom_point(
      aes(y = intelligibility),
      data = subset(
        data_fake_intelligibility,
        48 <= age_months & age_months <= 71
      )
    )
}

# Age of steepest growth for each centile
optimize_beta_gamlss_slope(
  model = m,
  centiles = c(5, 10, 50, 90),
  interval = range(data_fake_intelligibility$age_months)
)

# Manual approach: Make fine grid of predictions and find largest jump
centiles_grid <- predict_beta_gamlss(
  newdata = data.frame(age_months = seq(28, 95, length.out = 1000)),
  model = m
)
centiles_grid[which.max(diff(centiles_grid$c5)), "age_months"]

# When do children in different centiles reach 50%, 70% intelligibility?
uniroot_beta_gamlss(
  model = m,
  centiles = c(5, 10, 50),
  targets = c(.5, .7)
)
```

`fit_gen_gamma_gamlss` *Fit a generalized gamma regression model (for speaking rate)*

Description

The function fits the same type of GAMLSS model as used in Mahr and colleagues (2021): A generalized gamma regression model (via `gamlss.dist::GG()`) with natural cubic splines on the mean (mu), scale (sigma), and shape (nu) of the distribution. This model is fitted using this package's `mem_gamlss()` wrapper function.

Usage

```
fit_gen_gamma_gamlss(
  data,
  var_x,
  var_y,
  df_mu = 3,
  df_sigma = 2,
```

```
  df_nu = 1,
  control = NULL
)

fit_gen_gamma_gamlss_se(
  data,
  name_x,
  name_y,
  df_mu = 3,
  df_sigma = 2,
  df_nu = 1,
  control = NULL
)

predict_gen_gamma_gamlss(newdata, model, centiles = c(5, 10, 50, 90, 95))
```

Arguments

`data`	a data frame
`var_x`, `var_y`	(unquoted) variable names giving the predictor variable (e.g., `age`) and outcome variable (.e.g, `rate`).
`df_mu`, `df_sigma`, `df_nu`	degrees of freedom. If `0` is used, the `splines::ns()` term is dropped from the model formula for the parameter.
`control`	a `gamlss::gamlss.control()` controller. Defaults to `NULL` which uses default settings, except for setting trace to `FALSE` to silence the output from gamlss.
`name_x`, `name_y`	quoted variable names giving the predictor variable (e.g., `"age"`) and outcome variable (.e.g, `"rate"`). These arguments apply to `fit_gen_gamma_gamlss_se()`.
`newdata`	a one-column dataframe for predictions
`model`	a model fitted by `fit_gen_gamma_gamlss()`
`centiles`	centiles to use for prediction. Defaults to `c(5, 10, 50, 90, 95)`.

Details

There are two versions of this function. The main version is `fit_gen_gamma_gamlss()`, and it works with unquoted column names (e.g., `age`). The alternative version is `fit_gen_gamma_gamlss_se()`; the final "se" stands for "Standard Evaluation". This designation means that the variable names must be given as strings (so, the quoted `"age"` instead of bare name `age`). This alternative version is necessary when we fit several models using parallel computing with `furrr::future_map()` (as when using bootstrap resampling).

`predict_centiles()` will work with this function, but it will likely throw a warning message. Therefore, `predict_gen_gamma_gamlss()` provides an alternative way to compute centiles from the model. This function manually computes the centiles instead of relying on `gamlss::centiles()`. The main difference is that new x values go through `splines::predict.ns()` and then these are multiplied by model coefficients.

Value

for `fit_gen_gamma_gamlss()` and `fit_gen_gamma_gamlss_se()`, a `mem_gamlss()`-fitted model. The `.user` data in the model includes degrees of freedom for each parameter and the `splines::ns()` basis for each parameter. For `predict_gen_gamma_gamlss()`, a dataframe containing the model predictions for mu, sigma, and nu, plus columns for each centile in `centiles`.

Source

Associated article: https://doi.org/10.1044/2021_JSLHR-21-00206

Examples

```
data_fake_rates

m <- fit_gen_gamma_gamlss(data_fake_rates, age_months, speaking_sps)

# using "qr" in summary() just to suppress a warning message
summary(m, type = "qr")

# Alternative interface
d <- data_fake_rates
m2 <- fit_gen_gamma_gamlss_se(
  data = d,
  name_x = "age_months",
  name_y = "speaking_sps"
)
coef(m2) == coef(m)

# how to use control to change gamlss() behavior
m_traced <- fit_gen_gamma_gamlss(
  data_fake_rates,
  age_months,
  speaking_sps,
  control = gamlss::gamlss.control(n.cyc = 15, trace = TRUE)
)

# The `.user` space includes the spline bases, so that we can make accurate
# predictions of new xs.
names(m$.user)

# predict log(mean) at 55 months:
log_mean_55 <- cbind(1, predict(m$.user$basis_mu, 55)) %*% coef(m)
log_mean_55
exp(log_mean_55)

# But predict_gen_gamma_gamlss() does this work for us and also provides
# centiles
new_ages <- data.frame(age_months = 48:71)
centiles <- predict_gen_gamma_gamlss(new_ages, m)
centiles
```

```
# Confirm that the manual prediction matches the automatic one
centiles[centiles$age_months == 55, "mu"]
exp(log_mean_55)

if(requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot(pivot_centiles_longer(centiles)) +
    aes(x = age_months, y = .value) +
    geom_line(aes(group = .centile, color = .centile_pair)) +
    geom_point(
      aes(y = speaking_sps),
      data = subset(
        data_fake_rates,
        48 <= age_months & age_months <= 71
      )
    )
}

# Example of 0-df splines
m <- fit_gen_gamma_gamlss(
  data_fake_rates,
  age_months,
  speaking_sps,
  df_mu = 0,
  df_sigma = 2,
  df_nu = 0
)
coef(m, what = "mu")
coef(m, what = "sigma")
coef(m, what = "nu")

# mu and nu fixed, c50 mostly locked in
predict_gen_gamma_gamlss(new_ages, m)[c(1, 9, 17, 24), ]
```

`fit_kmeans` *Run (scaled) k-means on a dataset.*

Description

Observations are `scale()`-ed before clustering.

Usage

```
fit_kmeans(data, k, vars, args_kmeans = list())
```

Arguments

`data`	a dataframe
`k`	number of clusters to create

vars	variable selection for clustering. Select multiple variables with `c()`, e.g., `c(x, y)`. The selection supports tidyselect semantics tidyselect::select_helpers, e.g., `c(x, starts_with("mean_")`.
args_kmeans	additional arguments passed to `stats::kmeans()`.

Details

Note that each variable is `scaled()` before clustering and then cluster means are unscaled to match the original data scale.

This function provides the original kmeans labels as `.kmeans_cluster` but other alternative labeling based on different sortings of the data. These are provided in order to deal with label-swapping in Bayesian models. See bootstrapping example below.

Value

the original `data` but augmented with additional columns for clustering details. including `.kmeans_cluster` (cluster number of each observation, as a factor) and `.kmeans_k` (selected number of clusters).

Cluster-level information is also included. For example, suppose that we cluster using the variable `x`. Then the output will have a column `.kmeans_x` giving the cluster mean for `x` and `.kmeans_rank_x` giving the cluster labels reordered using the cluster means for `x`. The column `.kmeans_sort` contains the cluster sorted using the first principal component of the scaled variables. All columns of cluster indices are a `factor()` so that they can be plotted as discrete variables.

Examples

```
data_kmeans <- fit_kmeans(mtcars, 3, c(mpg, wt, hp))

library(ggplot2)
ggplot(data_kmeans) +
  aes(x = wt, y = mpg) +
  geom_point(aes(color = .kmeans_cluster))

ggplot(data_kmeans) +
  aes(x = wt, y = mpg) +
  geom_point(aes(color = .kmeans_rank_wt))

# Example of label swapping
set.seed(123)
data_boots <- lapply(
  1:10,
  function(x) {
    rows <- sample(seq_len(nrow(mtcars)), replace = TRUE)
    data <- mtcars[rows, ]
    data$.bootstrap <- x
    data
  }
) |>
  lapply(fit_kmeans, k = 3, c(mpg, wt, hp)) |>
```

```
  dplyr::bind_rows() |>
  dplyr::select(.bootstrap, dplyr::starts_with(".kmeans_")) |>
  dplyr::distinct()

# Clusters start off in random locations and move to center, so the labels
# differ between model runs and across bootstraps.
ggplot(data_boots) +
  aes(x = .kmeans_wt, y = .kmeans_mpg) +
  geom_point(aes(color = .kmeans_cluster)) +
  labs(title = "k-means centers on 10 bootstraps")

# Labels sorted using first principal component
# so the labels are more consistent.
ggplot(data_boots) +
  aes(x = .kmeans_wt, y = .kmeans_mpg) +
  geom_point(aes(color = .kmeans_sort)) +
  labs(title = "k-means centers on 10 bootstraps")
```

`format_year_month_age` *Convert age in months to years;months*

Description

Convert age in months to years;months

Usage

```
format_year_month_age(x)
```

Arguments

`x`	a vector ages in months

Details

Ages of `NA` return `"NA;NA"`.

This format by default is not numerically ordered. This means that `c("2;0", "10;10", "10;9")` would sort as `c("10;10", "10;9", "2;0")`. The function `stringr::str_sort(..., numeric = TRUE)` will sort this vector correctly.

Value

ages in the years;months format

Examples

```
ages <- c(26, 58, 25, 67, 21, 59, 36, 43, 27, 49)
format_year_month_age(ages)
```

`impute_values_by_length` *Staged imputation*

Description

Impute missing data at different utterance lengths using successive linear models.

Usage

```
impute_values_by_length(
  data,
  var_y,
  var_length,
  id_cols = NULL,
  include_max_length = FALSE,
  data_train = NULL
)
```

Arguments

`data`	dataframe in which to impute missing value
`var_y`	bare name of the response variable for imputation
`var_length`	bare name of the length variable
`id_cols`	a selection of variable names that uniquely identify each group of related observations. For example, `c(child_id, age_months)`.
`include_max_length`	whether to use the maximum length value as a predictor in the imputation models. Defaults to `FALSE`.
`data_train`	(optional) dataframe used to train the imputation models. For example, we might have data from a reference group of children in `data_train` but a clinical population in `data`. If omitted, the dataframe in `data` is used to train the models. `data_train` can also be a function. In this case, it is applied to the `data` argument in order to derive (filter) a subset of the data for training.

Value

a dataframe with the additional columns `{var_y}_imputed` (the imputed value), `.max_{var_length}` with the highest value of `var_length` with observed data, and `{var_y}_imputation` for labeling whether observations were `"imputed"` or `"observed"`.

Background

In Hustad and colleagues (2020), we modeled intelligibility data in young children's speech. Children would hear an utterance and then they would repeat it. The utterances started at 2 words in length, then increased to 3 words in length, and so on in batches of 10 sentences, all the way to 7 words in length. There was a problem, however: Not all of the children could produce utterances at every length. Specifically, if a child could not reliably produced 5 utterances of a given length length, the task was halted. So given the nature of the task, if a child had produced 5-word utterances, they also produced 2–4-word utterances as well.

The length of the utterance probably influenced the outcome variable: Longer utterances have more words that might help a listener understand the sentence, for example. Therefore, it did not seem appropriate to ignore the missing values. We used the following two-step procedure (see the Supplemental Materials for more detail):

Other notes:

Remark about `data` and `data_train`: One might ask, *why shouldn't some children help train the data imputation models?* Let's consider a norm-referenced standardized testing scenario: We have a new participant (observations in `data`), and we want to know how they compare to their age peers (participants in `data_train`). By separating out `data_train` and fixing it to a reference group, we can apply the same adjustment/imputation procedure to all new participants.

References

Hustad, K. C., Mahr, T., Natzke, P. E. M., & Rathouz, P. J. (2020). Development of Speech Intelligibility Between 30 and 47 Months in Typically Developing Children: A Cross-Sectional Study of Growth. *Journal of Speech, Language, and Hearing Research*, *63*(6), 1675–1687. https://doi.org/10.1044/2020_JSLHR-20-00008

Hustad, K. C., Mahr, T., Natzke, P. E. M., & J. Rathouz, P. (2020). Supplemental Material S1 (Hustad et al., 2020). ASHA journals. https://doi.org/10.23641/asha.12330956.v1

Examples

```
set.seed(1)
fake_data <- tibble::tibble(
  child = c(
    "a", "a", "a", "a", "a",
    "b", "b", "b", "b", "b",
    "c", "c", "c", "c", "c",
    "e", "e", "e", "e", "e",
    "f", "f", "f", "f", "f",
    "g", "g", "g", "g", "g",
    "h", "h", "h", "h", "h",
    "i", "i", "i", "i", "i"
  ),
  level = c(1:5, 1:5, 1:5, 1:5, 1:5, 1:5, 1:5, 1:5),
  x = c(
    c(100, 110, 120, 130, 150) + c(-8, -5, 0, NA, NA),
    c(100, 110, 120, 130, 150) + c(6, 6, 4, NA, NA),
    c(100, 110, 120, 130, 150) + c(-5, -5, -2, 2, NA),
    c(100, 110, 120, 130, 150) + rbinom(5, 12, .5) - 6,
```

```
    c(100, 110, 120, 130, 150) + rbinom(5, 12, .5) - 6,
    c(100, 110, 120, 130, 150) + rbinom(5, 12, .5) - 6,
    c(100, 110, 120, 130, 150) + rbinom(5, 12, .5) - 6,
    c(100, 110, 120, 130, 150) + rbinom(5, 12, .5) - 6
  )
)
data_imputed <- impute_values_by_length(
  fake_data,
  x,
  level,
  id_cols = c(child),
  include_max_length = FALSE
)

if (requireNamespace("ggplot2")) {
  library(ggplot2)
  ggplot(data_imputed) +
    aes(x = level, y = x_imputed) +
    geom_line(aes(group = child)) +
    geom_point(aes(color = x_imputation))
}
```

`join_to_split` *Join data onto resampled IDs*

Description

Join data onto resampled IDs

Usage

```
join_to_split(x, y, by, validate = FALSE)
```

Arguments

`x`	an rset object created by `rsample::bootstraps()`
`y`	y dataframe with a column of the id values which was resampled to create `x`
`by`	the name of column in y with the data
`validate`	whether to validate the join by counting the number of rows associated with each id. Defaults to `FALSE`.

Value

the original rset object with its `x$data` updated to join with `y` and with the row numbers `x$in_id` updated to work on the expanded dataset.

Examples

```
library(dplyr)
data_trees <- tibble::as_tibble(datasets::Orange)

data_tree_ids <- distinct(data_trees, Tree)

# Resample ids
data_bootstraps <- data_tree_ids %>%
  rsample::bootstraps(times = 20) %>%
  rename(splits_id = splits) %>%
  # Attach data to resampled ids
  mutate(
    data_splits = splits_id %>% purrr::map(
      join_to_split,
      data_trees,
      by = "Tree",
      validate = TRUE
    )
  )

data_bootstraps
```

`logitnorm_mean` *Compute the mean of logit-normal distribution(s)*

Description

This function is a wrapper around `logitnorm::momentsLogitnorm()`.

Usage

```
logitnorm_mean(mu, sigma)
```

Arguments

`mu`	mean(s) on the logit scale
`sigma`	standard deviation(s) on the logit scale

Value

the means of the distributions

Examples

```
x <- logitnorm_mean(2, 1)
x

# compare to simulation
```

```
set.seed(100)
rnorm(1000, 2, 1) |> plogis() |> mean()
```

`mem_gamlss` *Fit a gamlss model but store user data*

Description

Think of it as a gamlss model with memories (mem. gamlss).

Usage

```
mem_gamlss(...)
```

Arguments

`...`	arguments passed to gamlss::gamlss()

Value

the fitted `model` object but updated to include user information in `model$.user`. Includes the dataset used to fit the model `model$.user$data`, the session info `model$.user$session_info` and the call used to fit the model `model$.user$call`. `model$call` is updated to match

---

`predict_centiles` *Predict and tidy centiles from a GAMLSS model*

Description

gamlss has trouble doing predictions without the original training data.

Usage

```
predict_centiles(newdata, model, centiles = c(5, 10, 50, 90, 95), ...)

pivot_centiles_longer(data)
```

Arguments

`newdata`	a one-column dataframe for predictions
`model`	a gamlss model prepared by `mem_gamlss()`
`centiles`	centiles to use for prediction. Defaults to `c(5, 10, 50, 90, 95)`.
`...`	arguments passed to `gamlss::centiles.pred()`
`data`	centile predictions to reshape for `pivot_centiles_longer()`

Value

a tibble with fitted centiles for `predict_centiles()` and a long-format tibble with one centile value per row in `pivot_centiles_longer()`

`tocs_item` *Extract the TOCS details from a string (usually a filename)*

Description

Extract the TOCS details from a string (usually a filename)

Usage

```
tocs_item(xs)

tocs_type(xs)

tocs_length(xs)
```

Arguments

`xs`	a character vector

Value

`tocs_item()` returns the substring with the TOCS item, `tocs_type()` returns whether the item is `"single-word"` or `"multiword"`, and `tocs_length()` returns the length of the TOCS item (i.e., the number of words).

Examples

```
x <- c(
  "XXv16s7T06.lab", "XXv15s5T06.TextGrid", "XXv13s3T10.WAV",
  "XXv18wT11.wav", "non-matching", "s2T01"
)
data.frame(
  x = x,
  item = tocs_item(x),
  type = tocs_type(x),
  length = tocs_length(x)
)
```

`trapezoid_auc` *Compute AUCs using the trapezoid method*

Description

Compute AUCs using the trapezoid method

Usage

```
trapezoid_auc(xs, ys)

partial_trapezoid_auc(xs, ys, xlim)
```

Arguments

`xs`, `ys`	x and y positions
`xlim`	two-element vector (a range) of the `xs` to sum over

Value

the area under the curve computed using the trapezoid method. For `partial_trapezoid_auc()`, the partial area under the curve is computed.

Examples

```
if (requireNamespace("rstanarm", quietly = TRUE)) {
  wells <- rstanarm::wells
  r <- pROC::roc(switch ~ arsenic, wells)
  pROC::auc(r)
  trapezoid_auc(r$specificities, r$sensitivities)

  pROC::auc(r, partial.auc = c(.9, 1), partial.auc.focus = "sp")
  partial_trapezoid_auc(r$specificities, r$sensitivities, c(.9, 1))

  pROC::auc(r, partial.auc = c(.9, 1), partial.auc.focus = "se")
  partial_trapezoid_auc(r$sensitivities, r$specificities, c(.9, 1))

  pROC::auc(r, partial.auc = c(.1, .9), partial.auc.focus = "sp")
  partial_trapezoid_auc(r$specificities, r$sensitivities, c(.1, .9))

  pROC::auc(r, partial.auc = c(.1, .9), partial.auc.focus = "se")
  partial_trapezoid_auc(r$sensitivities, r$specificities, c(.1, .9))
}
```

`weight_lengths_with_ordinal_model` *Weight utterance lengths by using an ordinal regression model*

Description

For each participant, we find their length of longest utterance. We predict this longest utterance length as a nonlinear function of some variable, and we compute the probability of reaching each utterance length at each value of the predictor variable. These probabilities are then normalized to provide weights for each utterance length.

Usage

```
weight_lengths_with_ordinal_model(
  data_train,
  var_length,
  var_x,
  id_cols,
  spline_df = 2,
  data_join = NULL
)
```

Arguments

`data_train`	dataframe used to train the ordinal model. `data_train` can also be a function. In this case, it is applied to the `data_join` argument in order to derive (filter) a subset of the data for training.
`var_length`	bare name of the length variable. For example, `tocs_level`.
`var_x`	bare name of the predictor variable. For example, `age_months`.
`id_cols`	a selection of variable names that uniquely identify each group of related observations. For example, `c(child_id, age_months)`.
`spline_df`	number of degrees of freedom to use for the ordinal regression model.
`data_join`	(optional) dataset to use join the weights onto. This feature is necessary because we want to train a dataset on the observed data but supply the weights to the dataset with missing values imputed.

Value

the probability and weights of each utterance length at each observed value of `var_x`. These are in the added columns `{var_length}_prob_reached` and `{var_length}_weight`, respectively.

References

Hustad, K. C., Mahr, T., Natzke, P. E. M., & Rathouz, P. J. (2020). Development of Speech Intelligibility Between 30 and 47 Months in Typically Developing Children: A Cross-Sectional Study of Growth. *Journal of Speech, Language, and Hearing Research*, *63*(6), 1675–1687. https://doi.org/10.1044/2020_JSLHR-20-00008

Hustad, K. C., Mahr, T., Natzke, P. E. M., & J. Rathouz, P. (2020). Supplemental Material S1 (Hustad et al., 2020). ASHA journals. https://doi.org/10.23641/asha.12330956.v1

Examples

```
data_weights <- weight_lengths_with_ordinal_model(
  data_example_intelligibility_by_length,
  tocs_level,
  age_months,
  child,
  spline_df = 2
)

if (requireNamespace("ggplot2")) {
  library(ggplot2)
  p1 <- ggplot(data_weights) +
    aes(x = age_months, y = tocs_level_prob_reached) +
    geom_line(aes(color = ordered(tocs_level)), linewidth = 1) +
    scale_color_ordinal(end = .85) +
    labs(y = "Prob. of reaching length", color = "Utterance length")
  print(p1)

  p2 <- p1 +
    aes(y = tocs_level_weight) +
    labs(y = "Weight of utterance length")
  print(p2)
}
```

Index