Package: gofreg (via r-universe)

November 4, 2024

Title Bootstrap-Based Goodness-of-Fit Tests for Parametric Regression

Version 1.0.0

Description Provides statistical methods to check if a parametric family of conditional density functions fits to some given dataset of covariates and response variables. Different test statistics can be used to determine the goodness-of-fit of the assumed model, see Andrews (1997) <doi:10.2307/2171880>, Bierens & Wang (2012) <doi:10.1017/S0266466611000168>, Dikta & Scheer (2021) <doi:10.1007/978-3-030-73480-0> and Kremling & Dikta (2024) <doi:10.48550/arXiv.2409.20262>. As proposed in these papers, the corresponding p-values are approximated using a parametric bootstrap method.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Imports checkmate, dplyr, ggplot2, R6, survival

URL https://github.com/gkremling/gofreg, https://gkremling.github.io/gofreg/

BugReports https://github.com/gkremling/gofreg/issues

VignetteBuilder knitr

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

RemoteUrl https://github.com/gkremling/gofreg

RemoteRef HEAD

RemoteSha 3b0973c3f839dc91ffd2f344d2080eb070a910fc

Contents

`CondKolmbXY` *Conditional Kolmogorov test statistic for the joint distribution of (beta^T X,Y)*

Description

This class inherits from TestStatistic and implements a function to calculate the test statistic (and x-y-values that can be used to plot the underlying process).

The process underlying the test statistic is defined by

$$\bar{\alpha}_n(s,t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \left(I_{\{Y_i \leq t\}} - F(t|\hat{\vartheta}_n, X_i) \right) I_{\{\hat{\beta}_n^T X_i \leq s\}}, \quad (s,t) \in R^2.$$

Super class

`gofreg::TestStatistic` -> `CondKolmbXY`

Methods

Public methods:

- `CondKolmbXY$calc_stat()`
- `CondKolmbXY$clone()`

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
CondKolmbXY$calc_stat(data, model)
```

Arguments:

`data` `data.frame()` with columns x and y containing the data

`model` ParamRegrModel to test for, already fitted to the data

Returns: The modified object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CondKolmbXY$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
data <- dplyr::tibble(x = x, y = y)

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts <- CondKolmbXY$new()
ts$calc_stat(data, model)
print(ts)
plot(ts)

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts2 <- CondKolmbXY$new()
ts2$calc_stat(data, model2)
print(ts2)
plot(ts2)
```

CondKolmXY *Conditional Kolmogorov test statistic for the joint distribution of (X,Y)*

Description

This class inherits from TestStatistic and implements a function to calculate the test statistic (and x-y-values that can be used to plot the underlying process).

The process underlying the test statistic is given in Andrews (1997) doi:10.2307/2171880 and defined by

$$\nu_n(x,y) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \left(I_{\{Y_i \leq y\}} - F(y|\hat{\vartheta}_n, X_i) \right) I_{\{X_i \leq x\}}, \quad (x,y) \in R^{p+1}.$$

Super class

gofreg::TestStatistic -> CondKolmXY

Methods

Public methods:

- CondKolmXY$calc_stat()
- CondKolmXY$clone()

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
CondKolmXY$calc_stat(data, model)
```

Arguments:

`data` `data.frame()` with columns x and y containing the data

`model` ParamRegrModel to test for, already fitted to the data

Returns: The modified object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CondKolmXY$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
data <- dplyr::tibble(x = x, y = y)

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts <- CondKolmXY$new()
ts$calc_stat(data, model)
print(ts)
plot(ts)

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts2 <- CondKolmXY$new()
ts2$calc_stat(data, model2)
print(ts2)
plot(ts2)
```

CondKolmY *Conditional Kolmogorov test statistic for the marginal distribution of Y*

Description

This class inherits from TestStatistic and implements a function to calculate the test statistic (and x-y-values that can be used to plot the underlying process).

The process underlying the test statistic is given in Kremling & Dikta (2024) https://arxiv.org/abs/2409.20262 and defined by

$$\tilde{\alpha}_n(t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \left(I_{\{Y_i \leq t\}} - F(t|\hat{\vartheta}_n, X_i) \right), \quad -\infty \leq t \leq \infty.$$

Super class

gofreg::TestStatistic -> CondKolmY

Methods

Public methods:

- CondKolmY$calc_stat()
- CondKolmY$clone()

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
CondKolmY$calc_stat(data, model)
```

Arguments:

`data` `data.frame()` with columns x and y containing the data

`model` ParamRegrModel to test for, already fitted to the data

Returns: The modified object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CondKolmY$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
data <- dplyr::tibble(x = x, y = y)

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts <- CondKolmY$new()
ts$calc_stat(data, model)
print(ts)
plot(ts)

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts2 <- CondKolmY$new()
ts2$calc_stat(data, model2)
print(ts2)
plot(ts2)
```

CondKolmY_RCM *Conditional Kolmogorov test statistic for the marginal distribution of Y under random censorship*

Description

This class inherits from TestStatistic and implements a function to calculate the test statistic (and x-y-values that can be used to plot the underlying process).

The process underlying the test statistic is defined by

$$\tilde{\alpha}_n^{KM}(t) = \sqrt{n}\left(\hat{F}_n^{KM}(t) - \frac{1}{n}\sum_{i=1}^{n} F(t|\hat{\vartheta}_n, X_i)\right), \quad -\infty \leq t \leq \infty.$$

Super class

gofreg::TestStatistic -> CondKolmY_RCM

Methods

Public methods:

- CondKolmY_RCM$calc_stat()
- CondKolmY_RCM$clone()

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
CondKolmY_RCM$calc_stat(data, model)
```

Arguments:

`data` `data.frame()` with columns x and y containing the data

`model` ParamRegrModel to test for, already fitted to the data

Returns: The modified object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CondKolmY_RCM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
c <- rnorm(n, mean(y)*1.2, sd(y)*0.5)
data <- dplyr::tibble(x = x, z = pmin(y,c), delta = as.numeric(y <= c))

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE, loglik = loglik_xzd)

# Print value of test statistic and plot corresponding process
ts <- CondKolmY_RCM$new()
ts$calc_stat(data, model)
print(ts)
plot(ts)

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE, loglik = loglik_xzd)

# Print value of test statistic and plot corresponding process
ts2 <- CondKolmY_RCM$new()
ts2$calc_stat(data, model2)
print(ts2)
plot(ts2)
```

ExpGLM *Generalized linear model with exponential distribution*

Description

This class represents a generalized linear model with exponential distribution. It inherits from GLM and implements its functions that, for example, evaluate the conditional density and distribution functions.

Super classes

gofreg::ParamRegrModel -> gofreg::GLM -> ExpGLM

Methods

Public methods:

- ExpGLM$fit()
- ExpGLM$f_yx()
- ExpGLM$F_yx()
- ExpGLM$F1_yx()

- ExpGLM$sample_yx()
- ExpGLM$clone()

Method `fit()`**:** Calculates the maximum likelihood estimator for the model parameters based on given data.

Usage:

```
ExpGLM$fit(
  data,
  params_init = private$params,
  loglik = loglik_xy,
  inplace = FALSE
)
```

Arguments:

`data` tibble containing the data to fit the model to

`params_init` initial value of the model parameters to use for the optimization (defaults to the fitted parameter values)

`loglik` `function(data, model, params)` defaults to `loglik_xy()`

`inplace` `logical`; if `TRUE`, default model parameters are set accordingly and parameter estimator is not returned

Returns: MLE of the model parameters for the given data, same shape as `params_init`

Method `f_yx()`**:** Evaluates the conditional density function.

Usage:

```
ExpGLM$f_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional density shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tag beta), defaults to the fitted parameter values

Returns: value(s) of the conditional density function, same shape as `t`

Method `F_yx()`**:** Evaluates the conditional distribution function.

Usage:

```
ExpGLM$F_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional distribution shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tag beta), defaults to the fitted parameter values

Returns: value(s) of the conditional distribution function, same shape as `t`

Method `F1_yx()`**:** Evaluates the conditional quantile function.

Usage:

```
ExpGLM$F1_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional quantile function shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tag beta), defaults to the fitted parameter values

Returns: value(s) of the conditional quantile function, same shape as `t`

Method `sample_yx()`**:** Generates a new sample of response variables with the same conditional distribution.

Usage:

```
ExpGLM$sample_yx(x, params = private$params)
```

Arguments:

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tag beta), defaults to the fitted parameter values

Returns: vector of sampled response variables, same length as `nrow(x)`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ExpGLM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Use the built-in cars dataset
x <- datasets::cars$speed
y <- datasets::cars$dist
data <- dplyr::tibble(x=x, y=y)

# Create an instance of ExpGLM
model <- ExpGLM$new()

# Fit an Exponential GLM to the cars dataset
model$fit(data, params_init = list(beta=3), inplace=TRUE)
params_opt <- model$get_params()

# Plot the resulting regression function
plot(datasets::cars)
abline(a = 0, b = params_opt$beta)

# Generate a sample for y for given x following the same distribution
x.new <- seq(min(x), max(x), by=2)
y.smpl <- model$sample_yx(x.new)
points(x.new, y.smpl, col="red")

# Evaluate the conditional density, distribution, quantile and regression
# function at given values
model$f_yx(y.smpl, x.new)
model$F_yx(y.smpl, x.new)
model$F1_yx(y.smpl, x.new)
y.pred <- model$mean_yx(x.new)
points(x.new, y.pred, col="blue")
```

GammaGLM *Generalized linear model with gamma distribution*

Description

This class represents a generalized linear model with Gamma distribution. It inherits from GLM and implements its functions that, for example, evaluate the conditional density and distribution functions.

Super classes

gofreg::ParamRegrModel -> gofreg::GLM -> GammaGLM

Methods

Public methods:

- GammaGLM$fit()
- GammaGLM$f_yx()
- GammaGLM$F_yx()
- GammaGLM$F1_yx()
- GammaGLM$sample_yx()
- GammaGLM$clone()

Method `fit()`**:** Calculates the maximum likelihood estimator for the model parameters based on given data.

Usage:

```
GammaGLM$fit(
  data,
  params_init = private$params,
  loglik = loglik_xy,
  inplace = FALSE
)
```

Arguments:

`data` tibble containing the data to fit the model to

`params_init` initial value of the model parameters to use for the optimization (defaults to the fitted parameter values)

`loglik` `function(data, model, params)` defaults to loglik_xy()

`inplace` `logical`; if `TRUE`, default model parameters are set accordingly and parameter estimator is not returned

Returns: MLE of the model parameters for the given data, same shape as `params_init`

Method `f_yx()`**:** Evaluates the conditional density function.

Usage:

```
GammaGLM$f_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional density shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional density function, same shape as `t`

Method `F_yx()`**:** Evaluates the conditional distribution function.

Usage:

```
GammaGLM$F_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional distribution shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional distribution function, same shape as `t`

Method `F1_yx()`**:** Evaluates the conditional quantile function.

Usage:

```
GammaGLM$F1_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional quantile function shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional quantile function, same shape as `t`

Method `sample_yx()`**:** Generates a new sample of response variables with the same conditional distribution.

Usage:

```
GammaGLM$sample_yx(x, params = private$params)
```

Arguments:

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: vector of sampled response variables, same length as `nrow(x)`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GammaGLM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Use the built-in cars dataset
x <- datasets::cars$speed
y <- datasets::cars$dist
data <- dplyr::tibble(x=x, y=y)

# Create an instance of GammaGLM
model <- GammaGLM$new()

# Fit an Gamma GLM to the cars dataset
model$fit(data, params_init = list(beta=3, shape=1), inplace=TRUE)
params_opt <- model$get_params()

# Plot the resulting regression function
plot(datasets::cars)
abline(a = 0, b = params_opt$beta)

# Generate a sample for y for given x following the same distribution
x.new <- seq(min(x), max(x), by=2)
y.smpl <- model$sample_yx(x.new)
points(x.new, y.smpl, col="red")

# Evaluate the conditional density, distribution, quantile and regression
# function at given values
model$f_yx(y.smpl, x.new)
model$F_yx(y.smpl, x.new)
model$F1_yx(y.smpl, x.new)
y.pred <- model$mean_yx(x.new)
points(x.new, y.pred, col="blue")
```

GLM *Generalized linear model (abstract class)*

Description

This class specializes ParamRegrModel. It is the abstract base class for parametric generalized linear model objects with specific distribution family such as NormalGLM and handles the (inverse) link function.

Super class

`gofreg::ParamRegrModel` -> GLM

Methods

Public methods:

- `GLM$new()`
- `GLM$mean_yx()`

- GLM$clone()

Method `new()`**:** Initialize an object of class GLM.

Usage:

```
GLM$new(linkinv = identity, params = NA)
```

Arguments:

`linkinv` inverse link function, defaults to identity function

`params` model parameters to use as default (optional)

Returns: a new instance of the class

Method `mean_yx()`**:** Evaluates the regression function or in other terms the expected value of Y given X=x.

Usage:

```
GLM$mean_yx(x, params = private$params)
```

Arguments:

`x` vector of covariates

`params` model parameters to use, defaults to the fitted parameter values

Returns: value of the regression function

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GLM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`GLM.new` *Create GLM object with specific distribution family*

Description

This constructor function can be used to create an instance of a parametric GLM with specific distribution family, returning a new object of NormalGLM, ExpGLM, WeibullGLM or GammaGLM, depending on the value of `distr`.

Usage

```
GLM.new(distr, linkinv = identity, params = NA)
```

Arguments

`distr`	distribution family
`linkinv`	inverse link function, defaults to identity function
`params`	model parameters to use as default (optional)

Value

a new instance of a GLM-subclass

Examples

```
model <- GLM.new(distr = "normal")
# see examples of GLM-subclasses (e.g. NormalGLM) for how to use such models
```

`GOFTest` *Goodness-of-fit test for parametric regression*

Description

This class implements functions to calculate the test statistic for the original data as well as the statistics for bootstrap samples. It also offers the possibility to compute the corresponding bootstrap p-value.

Methods

Public methods:

- `GOFTest$new()`
- `GOFTest$get_stat_orig()`
- `GOFTest$get_stats_boot()`
- `GOFTest$get_pvalue()`
- `GOFTest$plot_procs()`
- `GOFTest$clone()`

Method `new()`**:** Initialize an instance of class GOFTest.

Usage:

```
GOFTest$new(
  data,
  model_fitted,
  test_stat,
  nboot,
  resample = resample_param,
  loglik = loglik_xy
)
```

Arguments:

`data` `data.frame()` containing the data

`model_fitted` object of class ParamRegrModel with fitted parameters

`test_stat` object of class TestStatistic

`nboot` number of bootstrap iterations

`resample` `function(data, model)` used to resample data in bootstrap iterations, defaults to `resample_param()`

`loglik function(data, model, params)` negative log-likelihood function used to fit model to resampled data in bootstrap iterations, defaults to `loglik_xy()`

Returns: a new instance of the class

Method `get_stat_orig()`**:** Calculates the test statistic for the original data and model.

Usage:

```
GOFTest$get_stat_orig()
```

Returns: object of class TestStatistic

Method `get_stats_boot()`**:** Calculates the test statistics for the resampled data and corresponding models.

Usage:

```
GOFTest$get_stats_boot()
```

Returns: vector of length `nboot` containing objects of class TestStatistic

Method `get_pvalue()`**:** Calculates the bootstrap p-value for the given model.

Usage:

```
GOFTest$get_pvalue()
```

Returns: p-value for the null hypothesis that `y` was generated according to `model`

Method `plot_procs()`**:** Plots the processes underlying the bootstrap test statistics (gray) and the original test statistic (red)

Usage:

```
GOFTest$plot_procs(
  title = sprintf("Test Statistic: %s, p-value: %s", class(private$test_stat)[1],
    self$get_pvalue()),
  subtitle = ggplot2::waiver(),
  color_boot = "gray40",
  color_orig = "red",
  x_lab = "plot.x",
  y_lab = "plot.y"
)
```

Arguments:

`title` text to be displayed as title of the plot; defaults to "Test statistic: xxx, p-value: xxx"

`subtitle` text to be displayed as subtitle of the plot; default is no subtitle

`color_boot` color used to plot bootstrap test statistics; default is "red"

`color_orig` color used to plot original test statistic; default is "gray40"

`x_lab` label to use for the x-axis; default is "plot.x"

`y_lab` label to use for the y-axis; default is "plot.y"

Returns: The object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GOFTest$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
data <- dplyr::tibble(x = x, y = y)

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Calculate the bootstrap p-value and plot the corresponding processes
goftest <- GOFTest$new(data, model, test_stat = CondKolmY$new(), nboot = 10)
goftest$get_pvalue()
goftest$plot_procs()

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Calculate the bootstrap p-value and plot the corresponding processes
goftest2 <- GOFTest$new(data, model2, test_stat = CondKolmY$new(), nboot = 10)
goftest2$get_pvalue()
goftest2$plot_procs()
```

`loglik_xy` *Negative log-likelihood function for a parametric regression model*

Description

The log-likelihood function for a parametric regression model with data (x,y) is given by the sum of the logarithm of the conditional density of Y given X=x evaluated at y.

This function is one option that can be used to fit a ParamRegrModel. It returns the negative log-likelihood value in order for `optim()` to maximize (instead of minimize).

Usage

```
loglik_xy(data, model, params)
```

Arguments

`data`	`list()` with tags x and y containing the data
`model`	ParamRegrModel to use for the likelihood function
`params`	vector with model parameters to compute likelihood function for

Value

Value of the negative log-likelihood function

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
params.true <- list(beta = c(2,3), sd = 1)
y <- model$sample_yx(x, params = params.true)
data <- dplyr::tibble(x = x, y = y)

# Compute negative log likelihood for true parameters
loglik_xy(data, model, params.true)

# Compute negative log likelihood for wrong parameters (should be higher)
loglik_xy(data, model, params = list(beta = c(1,2), sd = 0.5))
```

---

loglik_xzd *Negative log-likelihood function for a parametric regression model under random censorship*

Description

The log-likelihood function for a parametric regression model under random censorship with data (x,z,delta) is given by the sum of the logarithm of the conditional density of Y given X=x evaluated at z if z was uncensored or the logarithm of the conditional survival of Y given X=x evaluated at z if z was censored.

This function is one option that can be used to fit a ParamRegrModel. It returns the negative log-likelihood value in order for `optim()` to maximize (instead of minimize).

Usage

```
loglik_xzd(data, model, params)
```

Arguments

`data`	`list()` with tags x, z and delta containing the data
`model`	ParamRegrModel to use for the likelihood function
`params`	vector with model parameters to compute likelihood function for

Value

Value of the negative log-likelihood function

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
params.true <- list(beta = c(2,3), sd = 1)
y <- model$sample_yx(x, params = params.true)
c <- rnorm(n, mean(y) * 1.2, sd(y) * 0.5)
data <- dplyr::tibble(x = x, z = pmin(y, c), delta = as.numeric(y <= c))

# Compute negative log likelihood for true parameters
loglik_xzd(data, model, params.true)

# Compute negative log likelihood for wrong parameters (should be higher)
loglik_xzd(data, model, params = list(beta = c(1,2), sd = 0.5))
```

MEP *Marked empirical process test statistic for a given GLM*

Description

This class inherits from TestStatistic and implements a function to calculate the test statistic (and x-y-values that can be used to plot the underlying process).

The process underlying the test statistic is given in Dikta & Scheer (2021) doi:10.1007/9783030-734800 and defined by

$$\bar{R}_n^1(u) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \left(Y_i - m(X_i, \hat{\beta}_n) \right) I_{\{\hat{\beta}_n X_i \leq u\}}, \quad -\infty \leq u \leq \infty.$$

Super class

`gofreg::TestStatistic` -> `MEP`

Methods

Public methods:

- `MEP$calc_stat()`
- `MEP$clone()`

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
MEP$calc_stat(data, model)
```

Arguments:

`data` `data.frame()` with columns x and y containing the data

`model` ParamRegrModel to test for

Returns: The modified object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MEP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
data <- dplyr::tibble(x = x, y = y)

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts <- MEP$new()
ts$calc_stat(data, model)
print(ts)
plot(ts)

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts2 <- MEP$new()
ts2$calc_stat(data, model2)
print(ts2)
plot(ts2)
```

`NegBinomGLM` *Generalized linear model with negative binomial distribution*

Description

This class represents a generalized linear model with negative binomial distribution. It inherits from GLM and implements its functions that, for example, evaluate the conditional density and distribution functions.

Super classes

`gofreg::ParamRegrModel` -> `gofreg::GLM` -> `NegBinomGLM`

Methods

Public methods:

- NegBinomGLM$fit()
- NegBinomGLM$f_yx()
- NegBinomGLM$F_yx()
- NegBinomGLM$F1_yx()
- NegBinomGLM$sample_yx()
- NegBinomGLM$clone()

Method `fit()`**:** Calculates the maximum likelihood estimator for the model parameters based on given data.

Usage:

```
NegBinomGLM$fit(
  data,
  params_init = private$params,
  loglik = loglik_xy,
  inplace = FALSE
)
```

Arguments:

`data` tibble containing the data to fit the model to

`params_init` initial value of the model parameters to use for the optimization (defaults to the fitted parameter values)

`loglik` `function(data, model, params)` defaults to `loglik_xy()`

`inplace` `logical`; if `TRUE`, default model parameters are set accordingly and parameter estimator is not returned

Returns: MLE of the model parameters for the given data, same shape as `params_init`

Method `f_yx()`**:** Evaluates the conditional density function.

Usage:

```
NegBinomGLM$f_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional density shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional density function, same shape as `t`

Method `F_yx()`**:** Evaluates the conditional distribution function.

Usage:

```
NegBinomGLM$F_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional distribution shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional distribution function, same shape as `t`

Method `F1_yx()`**:** Evaluates the conditional quantile function.

Usage:

```
NegBinomGLM$F1_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional quantile function shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional quantile function, same shape as `t`

Method `sample_yx()`**:** Generates a new sample of response variables with the same conditional distribution.

Usage:

```
NegBinomGLM$sample_yx(x, params = private$params)
```

Arguments:

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: vector of sampled response variables, same length as `nrow(x)`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NegBinomGLM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Use the built-in cars dataset
x <- datasets::cars$speed
y <- datasets::cars$dist
data <- dplyr::tibble(x=x, y=y)

# Create an instance of a NegBinomGLM
model <- NegBinomGLM$new()

# Fit a Negative Binomial GLM to the cars dataset
model$fit(data, params_init = list(beta=3, shape=2), inplace=TRUE)
params_opt <- model$get_params()

# Plot the resulting regression function
plot(datasets::cars)
```

```
abline(a = 0, b = params_opt$beta)

# Generate a sample for y for given x following the same distribution
x.new <- seq(min(x), max(x), by=2)
y.smpl <- model$sample_yx(x.new)
points(x.new, y.smpl, col="red")

# Evaluate the conditional density, distribution, quantile and regression
# function at given values
model$f_yx(y.smpl, x.new)
model$F_yx(y.smpl, x.new)
model$F1_yx(y.smpl, x.new)
y.pred <- model$mean_yx(x.new)
points(x.new, y.pred, col="blue")
```

NormalGLM *Generalized linear model with normal distribution*

Description

This class represents a generalized linear model with normal distribution. It inherits from GLM and implements its functions that, for example, evaluate the conditional density and distribution functions.

Super classes

gofreg::ParamRegrModel -> gofreg::GLM -> NormalGLM

Methods

Public methods:

- NormalGLM$fit()
- NormalGLM$f_yx()
- NormalGLM$F_yx()
- NormalGLM$F1_yx()
- NormalGLM$sample_yx()
- NormalGLM$clone()

Method `fit()`**:** Calculates the maximum likelihood estimator for the model parameters based on given data.

Usage:

```
NormalGLM$fit(
  data,
  params_init = private$params,
  loglik = loglik_xy,
  inplace = FALSE
)
```

Arguments:

`data` tibble containing the data to fit the model to

`params_init` initial value of the model parameters to use for the optimization (defaults to the fitted parameter values)

`loglik` `function(data, model, params)` defaults to `loglik_xy()`

`inplace` `logical`; if `TRUE`, default model parameters are set accordingly and parameter estimator is not returned

Returns: MLE of the model parameters for the given data, same shape as `params_init`

Method `f_yx()`**:** Evaluates the conditional density function.

Usage:

```
NormalGLM$f_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional density shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and sd), defaults to the fitted parameter values

Returns: value(s) of the conditional density function, same shape as `t`

Method `F_yx()`**:** Evaluates the conditional distribution function.

Usage:

```
NormalGLM$F_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional distribution shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and sd), defaults to the fitted parameter values

Returns: value(s) of the conditional distribution function, same shape as `t`

Method `F1_yx()`**:** Evaluates the conditional quantile function.

Usage:

```
NormalGLM$F1_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional quantile function shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and sd), defaults to the fitted parameter values

Returns: value(s) of the conditional quantile function, same shape as `t`

Method `sample_yx()`**:** Generates a new sample of response variables with the same conditional distribution.

Usage:

```
NormalGLM$sample_yx(x, params = private$params)
```

Arguments:

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and sd), defaults to the fitted parameter values

Returns: vector of sampled response variables, same length as `nrow(x)`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NormalGLM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Use the built-in cars dataset
x <- datasets::cars$speed
y <- datasets::cars$dist
data <- dplyr::tibble(x=x, y=y)

# Create an instance of a NormalGLM
model <- NormalGLM$new()

# Fit a Normal GLM to the cars dataset
model$fit(data, params_init = list(beta=3, sd=2), inplace=TRUE)
params_opt <- model$get_params()

# Plot the resulting regression function
plot(datasets::cars)
abline(a = 0, b = params_opt$beta)

# Generate a sample for y for given x following the same distribution
x.new <- seq(min(x), max(x), by=2)
y.smpl <- model$sample_yx(x.new)
points(x.new, y.smpl, col="red")

# Evaluate the conditional density, distribution, quantile and regression
# function at given values
model$f_yx(y.smpl, x.new)
model$F_yx(y.smpl, x.new)
model$F1_yx(y.smpl, x.new)
y.pred <- model$mean_yx(x.new)
points(x.new, y.pred, col="blue")
```

`ParamRegrModel` *Parametric regression model (abstract class)*

Description

This is the abstract base class for parametric regression model objects like NormalGLM.

Parametric regression models are built around the following key tasks:

- A method `fit()` to fit the model to given data, i.e. compute the MLE for the model parameters
- Methods `f_yx()`, `F_yx()` and `mean_yx()` to evaluate the conditional density, distribution and regression function
- A method `sample_yx()` to generate a random sample of response variables following the model given a vector of covariates

Methods

Public methods:

- `ParamRegrModel$set_params()`
- `ParamRegrModel$get_params()`
- `ParamRegrModel$fit()`
- `ParamRegrModel$f_yx()`
- `ParamRegrModel$F_yx()`
- `ParamRegrModel$F1_yx()`
- `ParamRegrModel$sample_yx()`
- `ParamRegrModel$mean_yx()`
- `ParamRegrModel$clone()`

Method `set_params()`**:** Set the value of the model parameters used as default for the class functions.

Usage:

```
ParamRegrModel$set_params(params)
```

Arguments:

`params` model parameters to use as default

Returns: The modified object (`self`), allowing for method chaining.

Method `get_params()`**:** Returns the value of the model parameters used as default for the class functions.

Usage:

```
ParamRegrModel$get_params()
```

Returns: model parameters used as default

Method `fit()`**:** Calculates the maximum likelihood estimator for the model parameters based on given data.

Usage:

```
ParamRegrModel$fit(data, params_init = private$params, loglik = loglik_xy)
```

Arguments:

`data` list containing the data to fit the model to

`params_init` initial value of the model parameters to use for the optimization (defaults to the fitted parameter values)

`loglik function(data, model, params)` defaults to `loglik_xy()`

Returns: MLE of the model parameters for the given data, same shape as `params_init`

Method `f_yx()`**:** Evaluates the conditional density function.

Usage:

```
ParamRegrModel$f_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional density shall be evaluated

`x` vector of covariates

`params` model parameters to use, defaults to the fitted parameter values

Returns: value(s) of the conditional density function, same shape as `t`

Method `F_yx()`**:** Evaluates the conditional distribution function.

Usage:

```
ParamRegrModel$F_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional distribution shall be evaluated

`x` vector of covariates

`params` model parameters to use, defaults to the fitted parameter values

Returns: value(s) of the conditional distribution function, same shape as `t`

Method `F1_yx()`**:** Evaluates the conditional quantile function.

Usage:

```
ParamRegrModel$F1_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional quantile function shall be evaluated

`x` vector of covariates

`params` model parameters to use, defaults to the fitted parameter values

Returns: value(s) of the conditional quantile function, same shape as `t`

Method `sample_yx()`**:** Generates a new sample of response variables with the same conditional distribution.

Usage:

```
ParamRegrModel$sample_yx(x, params = private$params)
```

Arguments:

`x` vector of covariates

`params` model parameters to use, defaults to the fitted parameter values

Returns: vector of sampled response variables, same length as `x`

Method `mean_yx()`**:** Evaluates the regression function or in other terms the expected value of Y given X=x.

Usage:

```
ParamRegrModel$mean_yx(x, params = private$params)
```

Arguments:

`x` vector of covariates

`params` model parameters to use, defaults to the fitted parameter values

Returns: value of the regression function

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ParamRegrModel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`resample_param` *Parametric resampling scheme for a parametric regression model*

Description

Generate a new, resampled dataset of the same shape as data following the given model. The covariates are kept the same and the response variables are drawn according to `model$sample_yx()`.

Usage

```
resample_param(data, model)
```

Arguments

`data`	`data.frame()` with columns x and y containing the original data
`model`	ParamRegrModel to use for the resampling

Value

`data.frame()` with columns x and y containing the resampled data

Examples

```
# Create an example dataset
n <- 10
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
params <- list(beta = c(2, 3), sd = 1)
y <- model$sample_yx(x, params = params)
data <- dplyr::tibble(x = x, y = y)

# Fit the model to the data
model$fit(data, params_init = params, inplace = TRUE)

# Resample from the model given data
resample_param(data, model)
```

---

resample_param_cens *Parametric resampling scheme for a parametric regression model under random censorship*

Description

Generate a new, resampled dataset of the same shape as data following the given model. The covariates X are kept the same. Survival times Y are drawn according to `model$sample_yx()` and censoring times C according to the KM estimator.

Usage

```
resample_param_cens(data, model)
```

Arguments

`data`	`data.frame()` with columns x, z and delta containing the original data
`model`	ParamRegrModel to use for the resampling

Value

`data.frame()` with columns x, z and delta containing the resampled data

Examples

```
# Create an example dataset
n <- 10
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
params <- list(beta = c(2, 3), sd = 1)
y <- model$sample_yx(x, params = params)
c <- rnorm(n, mean(y) * 1.2, sd(y) * 0.5)
z <- pmin(y, c)
delta <- as.numeric(y <= c)
```

```
data <- dplyr::tibble(x = x, z = z, delta = delta)

# Fit the model to the data
model$fit(data, params_init = params, inplace = TRUE, loglik = loglik_xzd)

# Resample from the model given data
resample_param_cens(data, model)
```

`resample_param_rsmplx` *Parametric resampling scheme for a parametric regression model with resampling of covariates*

Description

Generate a new, resampled dataset of the same shape as data following the given model. The covariates are resampled from `data$x` and the response variables are drawn according to `model$sample_yx()`.

Usage

```
resample_param_rsmplx(data, model)
```

Arguments

`data`	`data.frame()` with columns x and y containing the original data
`model`	ParamRegrModel to use for the resampling

Value

`data.frame()` with columns x and y containing the resampled data

Examples

```
# Create an example dataset
n <- 10
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
params <- list(beta = c(2, 3), sd = 1)
y <- model$sample_yx(x, params = params)
data <- dplyr::tibble(x = x, y = y)

# Fit the model to the data
model$fit(data, params_init = params, inplace = TRUE)

# Resample from the model given data
resample_param(data, model)
```

SICM *Simulated integrated conditional moment test statistic*

Description

This class inherits from TestStatistic and implements a function to calculate the test statistic (and x-y-values that can be used to plot the underlying process).

The process underlying the test statistic is given in Bierens & Wang (2012) doi:10.1017/S0266466611000168 and defined by

$$\hat{T}_n^{(s)}(c) = \frac{1}{(2c)^{p+1}} \int_{[-c,c]^p} \int_{-c}^{c} \left| \frac{1}{\sqrt{n}} \sum_{j=1}^{n} \left(\exp(i\tau Y_j) - \exp(i\tau \tilde{Y}_j) \right) \exp(i\xi^T X_j) \right|^2 d\tau d\xi$$

Super class

gofreg::TestStatistic -> SICM

Methods

Public methods:

- SICM$new()
- SICM$calc_stat()
- SICM$clone()

Method new(): Initialize an instance of class SICM.

Usage:

```
SICM$new(
  c,
  transx = function(values) {
     tvals <- atan(scale(values))
     tvals[,
    apply(values, 2, sd) == 0] <- 0
     return(tvals)
 },
  transy = function(values, data) {
     array(atan(scale(values, center = mean(data$y),
    scale = sd(data$y))))
 }
)
```

Arguments:

c chosen value for integral boundaries (see Bierens & Wang (2012))

transx function(values) used to transform x-values to be standardized and bounded; default is standardization by subtracting the mean and dividing by the standard deviation and then applying arctan

`transy function(values, data)` used to transform y-values to be standardized and bounded (same method is used for simulated y-values); default is standardization by subtracting the mean and dividing by the standard deviation and then applying arctan

Returns: a new instance of the class

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
SICM$calc_stat(data, model)
```

Arguments:

`data data.frame()` with columns x and y containing the data

`model` ParamRegrModel to test for

Returns: The modified object (`self`), allowing for method chaining.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SICM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an example dataset
n <- 100
x <- cbind(runif(n), rbinom(n, 1, 0.5))
model <- NormalGLM$new()
y <- model$sample_yx(x, params=list(beta=c(2,3), sd=1))
data <- dplyr::tibble(x = x, y = y)

# Fit the correct model
model$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts <- SICM$new(c = 5)
ts$calc_stat(data, model)
print(ts)
plot(ts)

# Fit a wrong model
model2 <- NormalGLM$new(linkinv = function(u) {u+10})
model2$fit(data, params_init=list(beta=c(1,1), sd=3), inplace = TRUE)

# Print value of test statistic and plot corresponding process
ts2 <- SICM$new(c = 5)
ts2$calc_stat(data, model2)
print(ts2)
plot(ts2)
```

`TestStatistic` *Test Statistic for parametric regression models (abstract class)*

Description

This is the abstract base class for test statistic objects like CondKolmY or MEP.

Test statistics are built around the key method `calc_stat()` which calculates the particular test statistic (and x-y-values that can be used to plot the underlying process).

Methods

Public methods:

- `TestStatistic$get_value()`
- `TestStatistic$calc_stat()`
- `TestStatistic$get_plot_xy()`
- `TestStatistic$print()`
- `TestStatistic$geom_ts_proc()`
- `TestStatistic$plot()`
- `TestStatistic$clone()`

Method `get_value()`**:** Returns the value of the test statistic.

Usage:

```
TestStatistic$get_value()
```

Returns: value of the test statistic

Method `calc_stat()`**:** Calculate the value of the test statistic for given data and a model to test for.

Usage:

```
TestStatistic$calc_stat(data, model)
```

Arguments:

`data` `list()` containing the data

`model` ParamRegrModel to test for

Returns: The modified object (`self`), allowing for method chaining.

Method `get_plot_xy()`**:** Returns vectors of x and y that can be used to plot the process corresponding to the test statistic.

Usage:

```
TestStatistic$get_plot_xy()
```

Returns: list with plot.x and plot.y being vectors of the same length

Method `print()`**:** Overrides the print-method for objects of type `TestStatistic` to only print its value.

Usage:

```
TestStatistic$print()
```

Returns: The object (`self`), allowing for method chaining.

Method `geom_ts_proc()`**:** Creates a line plot showing the underlying process of the test statistic.

Usage:

```
TestStatistic$geom_ts_proc(...)
```

Arguments:

`...` Other arguments passed on to `ggplot2::geom_line()`. These are often aesthetics, used to set an aesthetic to a fixed value, like `colour = "red"` or `size = 3`.

Returns: A ggplot2 layer representing a line plot.

Method `plot()`**:** Creates a new ggplot showing the underlying process of the test statistic.

Usage:

```
TestStatistic$plot(...)
```

Arguments:

`...` Other arguments passed on to `ggplot2::geom_line()`. These are often aesthetics, used to set an aesthetic to a fixed value, like `colour = "red"` or `size = 3`.

Returns: A ggplot2 object representing the complete plot, including a line geometry.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TestStatistic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`WeibullGLM` *Generalized linear model with Weibull distribution*

Description

This class represents a generalized linear model with Weibull distribution. It inherits from GLM and implements its functions that, for example, evaluate the conditional density and distribution functions.

Super classes

`gofreg::ParamRegrModel` -> `gofreg::GLM` -> `WeibullGLM`

Methods

Public methods:

- WeibullGLM$fit()
- WeibullGLM$f_yx()
- WeibullGLM$F_yx()
- WeibullGLM$F1_yx()
- WeibullGLM$sample_yx()
- WeibullGLM$clone()

Method `fit()`**:** Calculates the maximum likelihood estimator for the model parameters based on given data.

Usage:

```
WeibullGLM$fit(
  data,
  params_init = private$params,
  loglik = loglik_xy,
  inplace = FALSE
)
```

Arguments:

`data` tibble containing the data to fit the model to

`params_init` initial value of the model parameters to use for the optimization (defaults to the fitted parameter values)

`loglik` `function(data, model, params)` defaults to `loglik_xy()`

`inplace` `logical`; if `TRUE`, default model parameters are set accordingly and parameter estimator is not returned

Returns: MLE of the model parameters for the given data, same shape as `params_init`

Method `f_yx()`**:** Evaluates the conditional density function.

Usage:

```
WeibullGLM$f_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional density shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional density function, same shape as `t`

Method `F_yx()`**:** Evaluates the conditional distribution function.

Usage:

```
WeibullGLM$F_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional distribution shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional distribution function, same shape as `t`

Method `F1_yx()`**:** Evaluates the conditional quantile function.

Usage:

```
WeibullGLM$F1_yx(t, x, params = private$params)
```

Arguments:

`t` value(s) at which the conditional quantile function shall be evaluated

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: value(s) of the conditional quantile function, same shape as `t`

Method `sample_yx()`**:** Generates a new sample of response variables with the same conditional distribution.

Usage:

```
WeibullGLM$sample_yx(x, params = private$params)
```

Arguments:

`x` matrix of covariates, each row representing one sample

`params` model parameters to use (`list()` with tags beta and shape), defaults to the fitted parameter values

Returns: vector of sampled response variables, same length as `nrow(x)`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
WeibullGLM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Use the built-in cars dataset
x <- datasets::cars$speed
y <- datasets::cars$dist
data <- dplyr::tibble(x=x, y=y)

# Create an instance of WeibullGLM
model <- WeibullGLM$new()

# Fit an Weibull GLM to the cars dataset
model$fit(data, params_init = list(beta=3, shape=1), inplace=TRUE)
params_opt <- model$get_params()

# Plot the resulting regression function
plot(datasets::cars)
```

```
abline(a = 0, b = params_opt$beta)

# Generate a sample for y for given x following the same distribution
x.new <- seq(min(x), max(x), by=2)
y.smpl <- model$sample_yx(x.new)
points(x.new, y.smpl, col="red")

# Evaluate the conditional density, distribution, quantile and regression
# function at given values
model$f_yx(y.smpl, x.new)
model$F_yx(y.smpl, x.new)
model$F1_yx(y.smpl, x.new)
y.pred <- model$mean_yx(x.new)
points(x.new, y.pred, col="blue")
```

Index