The Functions in the Package `gamlss.prepdata`

De Bastiani, F. Heller, G. Kneib, T. Mayr, A.
Rigby, R. A. Stasinopoulos, M. D. Stauffer, Reto
Umlauf, N, Zeileis, A.

Introduction

This booklet introduces the `gamlss.prepdata` package and its functionality. It aims to describe the available functions and how they can be used.

The latest versions of the packages **gamlss**, **gamlss2** and **gamlss.prepdata** are shown below:

```
rm(list=ls())
library(gamlss)
library(gamlss2)
library(ggplot2)
library(gamlss.ggplots)
library(gamlss.prepdata)
library("dplyr")
packageVersion("gamlss")
```

```
[1] '5.4.23'
```

```
packageVersion("gamlss2")
```

```
[1] '0.1.0'
```

```
packageVersion("gamlss.prepdata")
```

```
[1] '0.1.8'
```

gamlss.prepdata

The gamlss.prepdata package originated from the gamlss.ggplots package. As gamlss.ggplots became too large for easy maintenance, it was split into two separate packages, and gamlss.prepdata was created.

Since gamlss.prepdata is still at an experimental stage, some functions are hidden to allow time for thorough checking and validation. These hidden functions can still be accessed using the triple colon notation, for example: gamlss.prepdata:::.

The functions available in gamlss.prepdata are intended for pre-fitting — that is, to be used before applying the gamlss() or gamlss2() fitting functions. The available functions can be grouped into the following categories:

Information functions

These functions provide information about:

- The size of the dataset

- The presence and extent of missing values

- The structure of the dataset

  - Whether the variables are numeric or factors, and how they should be prepared for analysis

Plotting functions

These functions allow plotting for:

- individual variables

- Pairwise relationships between variables.

Features functions

Functions that assist in

- Detecting outliers
- Applying transformations
- Scaling variables.

Data Partition functions

Functions that facilitate partitioning data to improve inference and avoid overfitting during model selection.

Purpose and Usage

The information and plotting functions provide valuable insights that assist in building better models, including:

- Understanding the distribution of the *response* variable
- Choosing the appropriate *type* of analysis
- Examining explanatory variables, including:
  - Range and spread of values
  - Presence of missing values
  - *Associations* and *interactions* between explanatory variables
  - Nature of relationships between response and explanatory variables (*linear* or *non-linear*)

The features functions focus on handling outliers, scaling, and transforming explanatory variables (x-variables) before modeling.

Data partitioning is used to avoid overfitting by ensuring models are evaluated more reliably. It falls under the broader category of data manipulation, although merging datasets is not covered here — only partitioning for improved inference is addressed.

Most of the pre-fitting functions are data-related, and their names typically start with data (e.g., `data_NAME`), indicating that they either print information, produce plots, or manipulate `data.frame`s.

The `gamlss.prepdata` package is thus a useful tool for carrying out pre-analysis work before beginning the process of fitting a distributional regression model.

Next, we define what we mean by **distributional regression**.

Distributional Regression

The aim of this vignette is to demonstrate how to manipulate and prepare data before applying a distributional regression analysis.

The general form a distributional regression model can be written as;

$$\begin{aligned}\mathbf{y} &\overset{ind}{\sim} \mathcal{D}(\theta_1, \ldots, \theta_k)\\ g_1(\theta_1) &= \mathcal{ML}_1(\mathbf{x}_{11}, \mathbf{x}_{21}, \ldots, \mathbf{x}_{p1})\\ \ldots &= \ldots\\ g_k(\theta_k) &= \mathcal{ML}_k(\mathbf{x}_{1k}, \mathbf{x}_{2k}, \ldots, \mathbf{x}_{pk}).\end{aligned}$$

where we assume that the response variable y_i for $i = 1, \ldots, n$, is independently distributed having a distribution $\mathcal{D}(\theta_1, \ldots, \theta_k)$ with k parameters and where all parameters could be effected by the explanatory variables $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_p$. The $\mathcal{ML}$ represents **any** regression type machine learning algorithm i.e. LASSO, Neural networks etc.

When only **additive smoothing** terms are used in the fitting the model can be written as;

$$\begin{aligned}\mathbf{y} &\overset{ind}{\sim} \mathcal{D}(\theta_1, \ldots, \theta_k)\\ g_1(\theta_1) &= b_{01} + s_1(\mathbf{x}_{11}) + \cdots, + s_p(\mathbf{x}_{p1})\\ \ldots &= \ldots\\ g_k(\theta_k) &= b_{0k} + s_1(\mathbf{x}_{1k}) + \cdots, + s_p(\mathbf{x}_{pk}).\end{aligned}$$

which is the GAMLSS model introduced by Rigby and Stasinopoulos (2005).

There are three books on GAMLSS, D. M. Stasinopoulos et al. (2017), Rigby et al. (2019) and M. D. Stasinopoulos et al. (2024) and several 'GAMLSS lecture materials, available from GitHUb, https://github.com/mstasinopoulos/Porto_short_course.git and https://github.com/mstasinopoulos/ShortCourse.git. The latest **R** packages related to GAMLSS can be found in https://gamlss-dev.r-universe.dev/builds.

The aim of this package is to prepare data and extract useful information that can be utilized during the modeling stage.

Information functions

The functions for obtaining information from a dataset are described in this section and summarized in Table 1. These functions can provide:

- General information about the dataset such as the dimensions (number of rows and columns) and the percentage of omitted (missing) observations
- Information about the variables in the dataset
- Information about the observations in the dataset

All functions have a `data.frame` as their first argument.

Here is a table of the information functions;

Table 1: A summary table of the functions to obtain information

Functions	Usage
`data_dim()`	Returns the number of rows and columns, and calculates the percentage of omitted (missing) observations
`data_names()`	Lists the names of the variables in the dataset
`data_rename()`	Allows renaming of one or more variables in the dataset
`data_shorter_names()`	Allows shortening the names of one or more variables in the dataset
`data_distinct()`	Displays the number of distinct values for each variable in the dataset
`data_which_na()`	Displays the count of NA (missing) values for each variable in the dataset
`data_omit()`	Removes all rows (observations) that contain any NA (missing) values
`data_str()`	Displays the class of each variable (e.g., numeric, factor, etc.) along with additional details about the variables
`data_cha2fac()`	Converts all character variables in the dataset to factor type
`data_few2fac()`	Converts variables with a small number of distinct values (e.g., binary or categorical) to factor type
`data_int2num()`	Converts integer variables with several distinct values to numeric type to allow for continuous analysis
`data_rm()`	Removes one or more variables (columns) from the dataset as specified by the user
`data_rmNAvars()`	Removes variables which have NA's as all its values.
`data_rm1val()`	Removes factors that have only a single level (no variability) in the dataset
`data_select()`	Allows the user to select one or more specific variables (columns) from the dataset for further analysis

Functions	Usage
data_exclude_class()	Removes all variables of a specified class (e.g., factor, numeric) from the dataset
data_only_continuous()	Retains only the numeric variables (columns) in the dataset, excluding factors or other variable types

Next we give examples of the functions.

data_dim()

This function provides detailed information about the dimensions of a `data.frame`. It is similar to the R function `dim()`, but with additional details. The output is the original data frame, allowing it to be used in a series of piping commands.

```
rent99 |> data_dim()
```

```
************************************************************
************************************************************
the R class of the data is: data.frame
the dimensions of the data are: 3082 by 9
number of observations with missing values: 0
% of NA's in the data: 0 %
************************************************************
************************************************************
```

back to table

data_names()

This function provides the names of the variables in the `data.frame`, similar to the `R` function `names()`.

```
rent99 |> data_names()
```

```
************************************************************
************************************************************
the names of variables
[1] "rent"     "rentsqm"  "area"     "yearc"    "location" "bath"     "kitchen"
```

```
[8] "cheating" "district"
************************************************************
************************************************************
```

The output of the function is the original data frame. back to table

data_rename()

Renames one or more variables (columns) in the `data.frame` using the function `data_rename()`;

```
da<- rent99 |> data_rename(oldname="rent", newname="R")
head(da)
```

	R	rentsqm	area	yearc	location	bath	kitchen	cheating	district
1	109.9487	4.228797	26	1918	2	0	0	0	916
2	243.2820	8.688646	28	1918	2	0	0	1	813
3	261.6410	8.721369	30	1918	1	0	0	1	611
4	106.4103	3.547009	30	1918	2	0	0	0	2025
5	133.3846	4.446154	30	1918	2	0	0	1	561
6	339.0256	11.300851	30	1918	2	0	0	1	541

The output of the function is the original data frame with new names.

back to table

data_shorter_names()

If the variables in the dataset have very long names, they can be difficult to handle in formulae during modelling. The function `data_shorter_names()` abbreviates the names of the explanatory variables, making them easier to use in formulas.

```
rent99 |> data_shorter_names()
```

```
************************************************************
************************************************************
the names of variables
[1] "rent"  "rents" "area"  "yearc" "locat" "bath"  "kitch" "cheat" "distr"
************************************************************
************************************************************
```

If no long variable names exist in the dataset, the function data_shorter_names() does nothing. However, when applicable, the function abbreviates long names and returns the original data frame with the new shortened names.

> **Warning**
>
> Note that there is a risk when using a small value for the `max` option, as it may result in identical names for different variables. This could lead to confusion or errors in the modelling process. It is important to carefully choose an appropriate value for `max` to avoid this issue.

back to table

data_distinct()

TThe distinct values for each variable are shown below.

```
rent99 |> data_distinct()
```

```
    rent  rentsqm     area    yearc location     bath  kitchen cheating
    2723     3053      132       68        3        2        2        2
district
     336
```

The output of the function is the original data frame.

back to table

data_which_na()

This function provides information about which variables have missing observations and how many missing values there are for each variable;

```
rent99 |> data_which_na()
```

```
    rent  rentsqm     area    yearc location     bath  kitchen cheating
       0        0        0        0        0        0        0        0
district
       0
```

The output of the function is the original data frame nothing is changing.

back to table

data_omit

The function `data_str()` (similar to the `R` function `str()`) provides information about the types of variables present in the dataset.

```
rent99 |> data_omit()
```

```
************************************************************
************************************************************
the R class of the data is: data.frame
the dimensions of the data before omition are: 3082 x 9
the dimensions of the data saved after omition are: 3082 x 9
the number of observations omited: 0
************************************************************
************************************************************
```

The output of the function is a new data frame with all NA's omitted.

> **Warning**
>
> It is important to select the relevant variables before using the data_omit() function, as some unwanted variables may contain many missing values that could lead to unnecessary row omissions.

back to table

data_str()

The function `data_str()` (similar to the **R** function `str()`) provides information about the types of variable exist in the data.

```
rent99 |> data_str()
```

```
************************************************************
************************************************************
the structure of the data
'data.frame':   3082 obs. of  9 variables:
 $ rent    : num  110 243 262 106 133 ...
 $ rentsqm : num  4.23 8.69 8.72 3.55 4.45 ...
 $ area    : int  26 28 30 30 30 30 31 31 32 33 ...
 $ yearc   : num  1918 1918 1918 1918 1918 ...
 $ location: Factor w/ 3 levels "1","2","3": 2 2 1 2 2 2 1 1 1 2 ...
```

```
 $ bath    : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
 $ kitchen : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 2 1 1 ...
 $ cheating: Factor w/ 2 levels "0","1": 1 2 2 1 2 2 1 2 1 1 ...
 $ district: int  916 813 611 2025 561 541 822 1713 1812 152 ...
************************************************************
************************************************************
table of the class of variabes

 factor integer numeric
      4       2       3
************************************************************
************************************************************
distinct values in variables
    rent  rentsqm     area    yearc location     bath  kitchen cheating
    2723     3053      132       68        3        2        2        2
district
     336
consider to make those characters vectors into factors:
location bath kitchen cheating
************************************************************
************************************************************
```

The output of the function is the original data frame.

The next set of functions manipulate variables withing `data.frame`s.

back to table

`data_cha2fac()`

Often, variables in datasets are read as character vectors, but for analysis, they may need to be treated as factors. This function transforms any character vector (with a relatively small number of distinct values) into a factor.

```
rent99 |> data_cha2fac()  -> da
```

```
************************************************************
not character vector was found
```

Since no character were found nothing have changed. The output of the function is a new data frame.

back to table

data_few2fac()

There are occasions when some variables have very few distinct observations, and it may be better to treat them as factors. The function `data_few2fac()` converts vectors with a small number of distinct values into factors.

```
rent99 |>  data_few2fac() -> da
```

```
************************************************************
    rent  rentsqm     area    yearc location     bath  kitchen cheating
    2723     3053      132       68        3        2        2        2
district
     336
************************************************************
4 vectors with fewer number of values than 5 were transformed to factors
************************************************************
************************************************************
```

```
str(da)
```

```
'data.frame':   3082 obs. of  9 variables:
 $ rent    : num  110 243 262 106 133 ...
 $ rentsqm : num  4.23 8.69 8.72 3.55 4.45 ...
 $ area    : int  26 28 30 30 30 30 31 31 32 33 ...
 $ yearc   : num  1918 1918 1918 1918 1918 ...
 $ location: Factor w/ 3 levels "1","2","3": 2 2 1 2 2 2 1 1 1 2 ...
 $ bath    : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
 $ kitchen : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 2 1 1 ...
 $ cheating: Factor w/ 2 levels "0","1": 1 2 2 1 2 2 1 2 1 1 ...
 $ district: int  916 813 611 2025 561 541 822 1713 1812 152 ...
```

The output of the function is a new data frame. This can be seen by using the **R** function `str()`;

back to table

data_int2num()

Occasionally, we need to convert integer variables with a very large range of values into numeric vectors, especially for graphics. The function `data_int2num()` performs this conversion.

```
rent99 |> data_int2num() -> da
```

```
************************************************************
    rent  rentsqm     area    yearc location     bath  kitchen cheating
    2723     3053      132       68        3        2        2        2
district
     336
2 integer vectors with more number of values than 50 were transformed to numeric
************************************************************
```

The output of the function is a new data frame.

back to table

data_rm()

For plotting datasets, it's easier to keep only the relevant variables. The function `data_rm()` allows you to remove unnecessary variables, making the dataset more manageable for visualization.

```
data_rm(rent99, c(2,9)) -> da
dim(rent99)
```

```
[1] 3082    9
```

```
dim(da)
```

```
[1] 3082    7
```

The output of the function is a new data frame. Note that this could be also done using the function `select()` of the package **dplyr**, or our own `data_select()` function.

back to table

data_rmNAvars()

Occutionally when we read data from files in Rsome extra variables are accidentaly produced with no values but NA's. The function `data_rmNAvars()` removes those variables from the data;

```
data_rmNAvars(da)
```

back to table

data_rm1val()

This function searches for variables with only a single distinct value (often factors left over from a previous `subset()` operation) and removes them from the dataset.

```
da |> data_rm1val()
```

The output of the function is a new data frame.

back to table

data_select()

This function selects specified variables from a dataset.

```
da1<-rent |> data_select( vars=c("R", "Fl", "A"))
head(da1)
```

```
       R Fl    A
1  693.3 50 1972
2  422.0 54 1972
3  736.6 70 1972
4  732.2 50 1972
5 1295.1 55 1893
6 1195.9 59 1893
```

The output of the function is a new data frame.

back to table

data_exclude_class()

This function searches for variables (columns) of a specified `R` class and removes them from the dataset. By default, the class to remove is factor.

```
da |> data_exclude_class()  -> da1
head(da1)
```

```
      rent area yearc
1 109.9487   26  1918
2 243.2820   28  1918
3 261.6410   30  1918
4 106.4103   30  1918
5 133.3846   30  1918
6 339.0256   30  1918
```

The output of the function is a new data frame.

data_only_continuous()

This function keeps only the continuous variables in the dataset, removing all non-continuous variables.

```
da |> data_only_continuous()  -> da1
head(da1)
```

```
      rent area yearc
1 109.9487   26  1918
2 243.2820   28  1918
3 261.6410   30  1918
4 106.4103   30  1918
5 133.3846   30  1918
6 339.0256   30  1918
```

The output of the function is a new data frame.

back to table

Graphical functions

Graphical methods are used in pre-analysis to examine individual variables or pair-wise relationships between variables.

Table 2: A summary table of the graphical functions

Functions	Usage
`data_plot()`	Plots each variable in the dataset to visualize its distribution or other important characteristics (see also Section
`data_bucket()`	Generates bucket plots for all numerical variables in the dataset to visualize their skewness and kurtosis
`data_response()`	Plots the response variable alongside its z-score, providing a standardized version of the response for comparison
`data_zscores()`	Plots the z-scores for all continuous variables in the dataset, allowing for easy visualization of the standardized values (see also Section)
`data_xyplot()`	Generates pairwise plots of the response variable against all other variables in the dataset to visualize their relationships
`data_cor()`	Calculates and plots the pairwise correlations for all continuous variables in the dataset to assess linear relationships between them
`data_void()`	Searches for pairwise empty spaces across all continuous variables in the dataset to identify problems with interpretation or prediction
`data_pcor()`	Calculates and plots the pairwise patrial-correlations for all continuous variables in the dataset
`data_inter()`	Searches for potential pairwise interactions between variables in the dataset to identify relationships or dependencies that may be useful for modelling
`data_leverage()`	Detects outliers in the continuous explanatory variables (x-variables) as a group to highlight unusual observations
`data_Ptrans_plot()`	Plots the response variable against various power transformations of the continuous x-variables to explore potential relationships and model suitability

Next we give examples of the graphical functions.

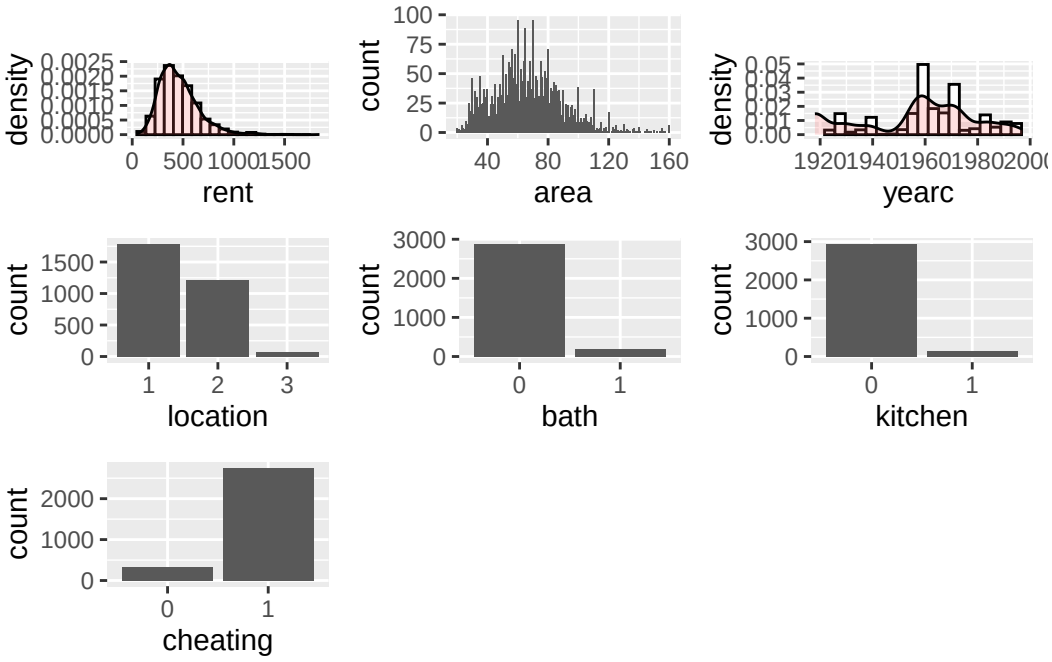
`data_plot()`

The function `data_plot` plots all the variables of the data individually; It plots the continuous variable as *histograms* with a density plots superimposed, see the plot for `rent` and `yearc`. As an alternative a `dot plots` can be requested see for an example in **?@sec-data_response**. For integers the function plots *needle* plots, see `area` below and for categorical the function plots *bar* plots, see `location`, `bath` `kitchen` and `cheating` below.

```
da |> data_plot()
```

```
 100 % of data are saved,
that is, 3082 observations.
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_bar()`).
Removed 2 rows containing missing values or values outside the scale range
(`geom_bar()`).
```



The function could saves the `ggplot2` figures.

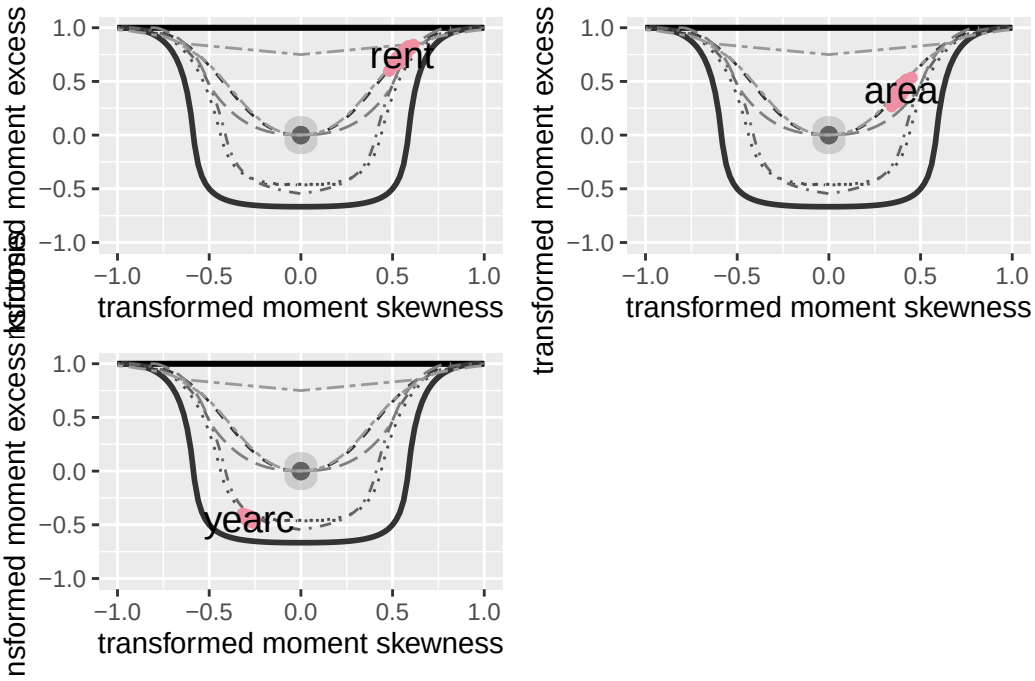
back to table

`data_bucket()`

The function `data_bucket` plots bucket plots (which identify skewness and kurosid) for all variables in the data set.

```
da |> data_bucket()
```

```
 100 % of data are saved,
that is, 3082 observations.
    rent     area    yearc location     bath  kitchen cheating
    2723      132       68        3        2        2        2
```



back to table

`data_response()`

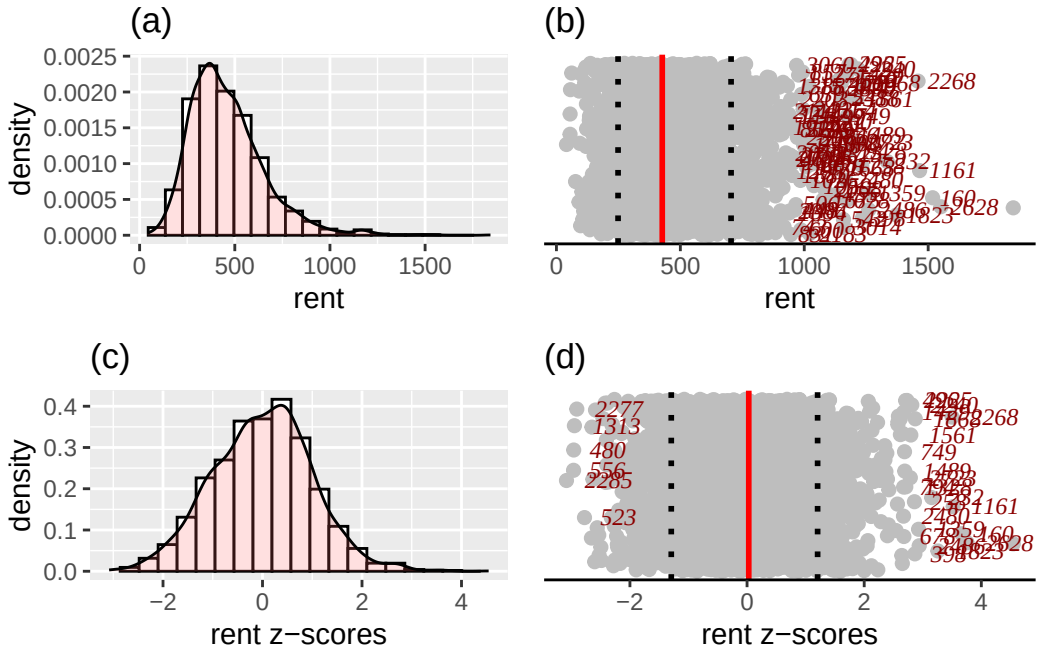
The function `data_response()` plots four different plots related to a continuous response variable. It plots; i) a histogram (and the density function); ii) a dot plot of the response in the original scale iii) a histogram (and the density function) of the response at the z-scores scale and vi) s dot-plot in the z-score scale.

The z-score scale is defined by fitting a distribution to the variable (normal or SHASH) and then taking the residuals, see also next section. The dot plots are good in identify highly skew

variables and unusual observations. They display the median and inter quantile range of the data. The y-axis of a dot plot is a randomised uniform variable (therefore the plot could look slightly different each time.)

```
da |> data_response(, response=rent)
```

```
 100 % of data are saved,
that is, 3082 observations.
the class of the response is numeric is this correct?
a continuous distribution on (0,inf) could be used
```

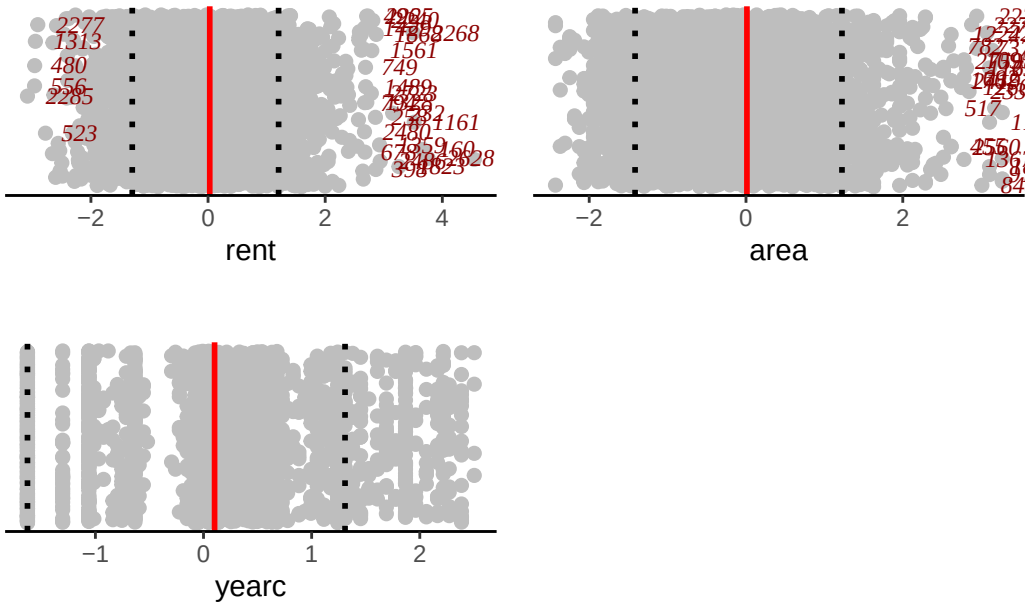


back to table

`data_zscores()`

One could fit any four parameter (GAMLSS) distribution, defined on $-\infty$ to ∞, to any continuous variable, where skewness and kurtosis is suspected and take the quantile residuals (or z-scores) as the transformed values x-values. The function `y_zscores()` performs just this. It takes a continuous variable and fits a continuous four parameter distribution and gives the z-scores. The fitting distribution can be specified by the user, but as default we use the SHASHo distribution. Below we demonstrate that the function y_zscores() is equivalent to fitting the SHASHo distribution to a continuous variable and then take the quqntile residuals from the fitted model as the result.

```
da |> data_zscores()
```



In order to see how the z-scores are calculated consider the function `y_zscores()` which is taking an individual variables z-scores;

```
 z <- y_zscores(rent99$rent, plot=FALSE)
```

The function is equivalent of fitting a constant model to all the parameters of a given distribution and than taking the quantile residuals (or z=scores) as he variable of interest. The default distribution is the four parameter `SHASHo` distribution.

```
library(gamlss2)
m1 <- gamlssML(rent99$rent, family=SHASHo) # fitting a 4 parameter distribution
cbind(z,resid(m1))[1:5,]#  and taking the residuals
```

```
          z
1 -2.496097 -2.496097
2 -1.340464 -1.340464
3 -1.182014 -1.182014
4 -2.526326 -2.526326
5 -2.295069 -2.295069
```

back to table

`data_xyplot()`

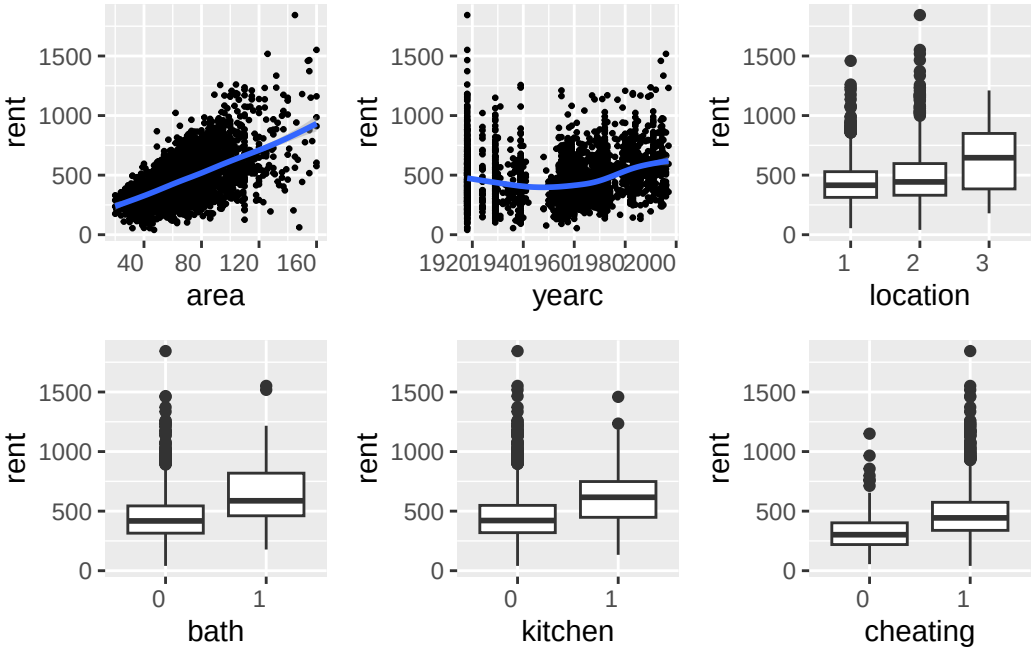
The functions `data_xyplot()` plots the response variable against each of the independent explanatory variables. It plots the continuous against continuous as *scatter* plots and continuous variables against categorical as *box* plot.

> **Note**
>
> At the moment there is no provision for categorical response variables.

```
da |> data_xyplot(response=rent )
```

```
 100 % of data are saved,
that is, 3082 observations.
```

```
`geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
`geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



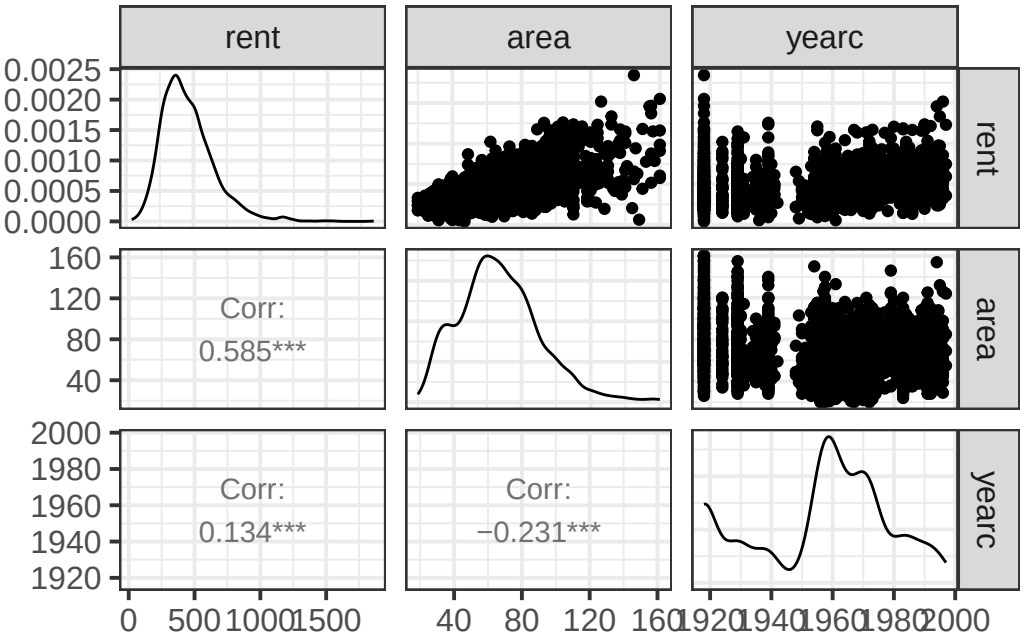
The output of the function saves the `ggplot2` figures.

Note hat the package `gamlss.prepdata` do not provides pairwise plot of the explanatory variables themself but the package `GGally` does. here is an example ;

```
library(GGally)
```

```
Registered S3 method overwritten by 'GGally':
  method from
  +.gg   ggplot2
```

```
dac <- gamlss.prepdata:::data_only_continuous(da)
ggpairs(dac, lower=list(continuous = "cor",
      combo = "box_no_facet", discrete = "count",
      na = "na"),upper=list(continuous = "points",
      combo = "box_no_facet", discrete = "count", na = "na")) +
theme_bw(base_size =15)
```



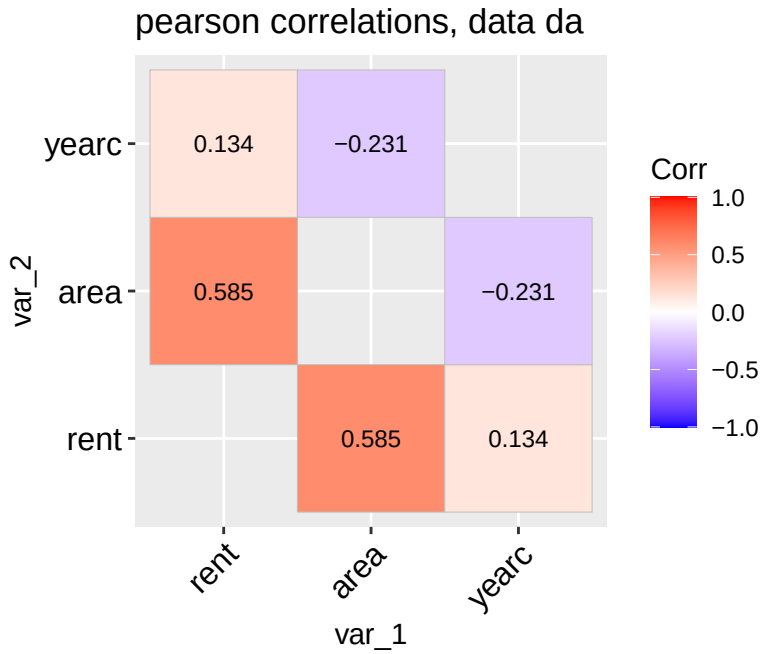
back to table

data_cor()

The function `data_corr()` is taking a `data.frame` object and plot the correlation coefficients of all its continuous variables.

```
data_cor(da, lab=TRUE)
```

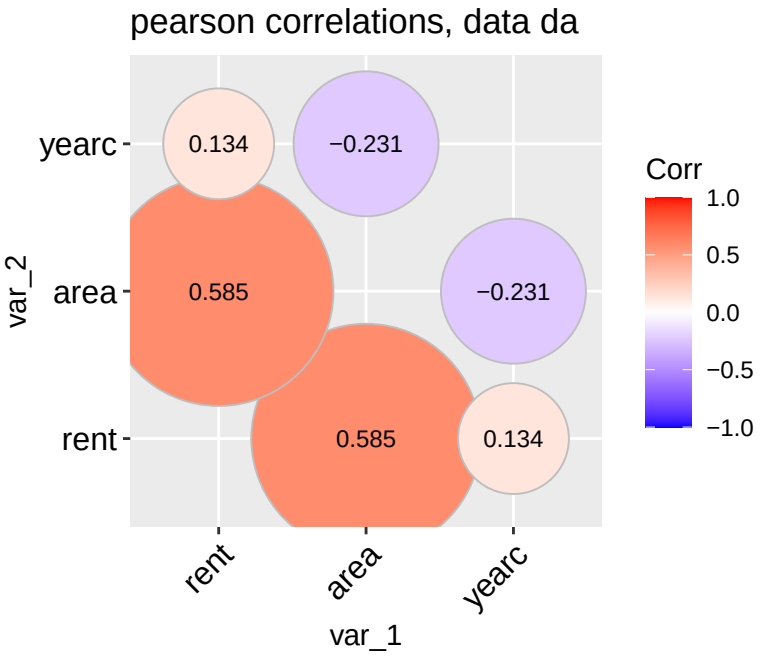
```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_cor(da, lab = TRUE):
```



A different type of plot can be produce if we use;

```
data_cor(da, method="circle", circle.size = 40)
```

```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_cor(da, method = "circle", circle.size = 40):
```



To get the variables with higher, say, than \$0.4 \$ correlation values use;

```
Tabcor <- data_cor(da, plot=FALSE)
```

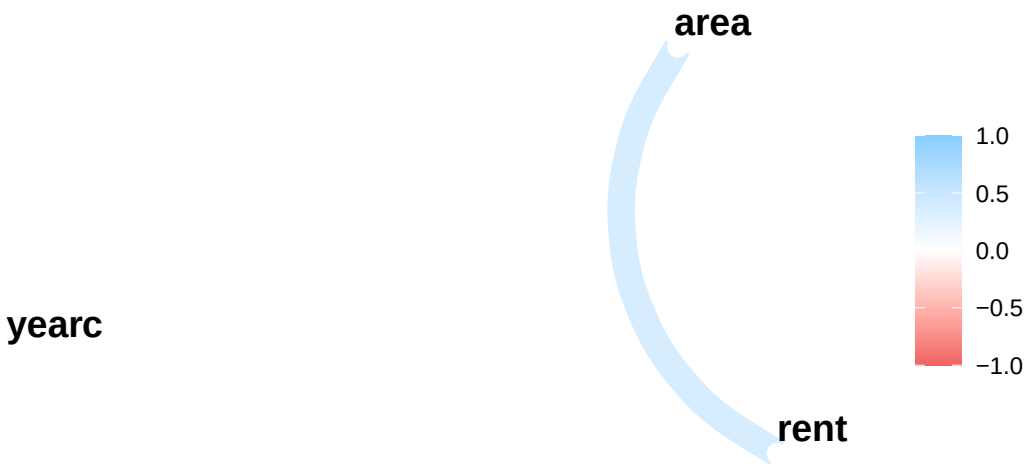
```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_cor(da, plot = FALSE):
```

```
high_val(Tabcor, val=0.4)
```

```
     name1  name2  corrs
[1,] "rent" "area" "0.585"
```

We can plot the path of those variables using the package `corr`;

```
library(corrr)
network_plot(Tabcor)
```



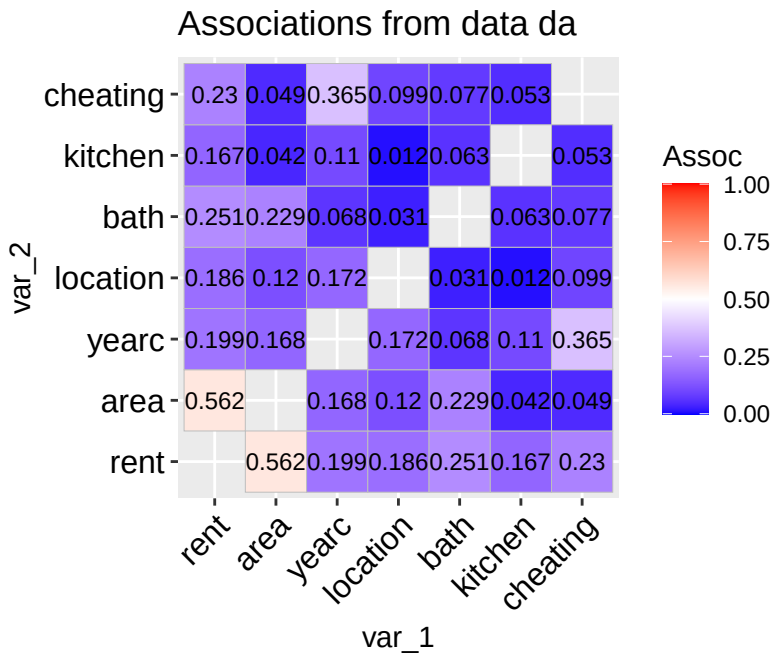
back to table

`data_association()`

The function `data_association()` is taking a `data.frame` object and plot the pair-wise association of all its variables. The pair-wise association for two continuous variables is given by default by the absolute value of the Spearman's correlation coefficient. For two categorical variables by the (adjusted for bias) Cramers' v-coefficient. For a continuous against a categorical variables by the $\sqrt{R}^2$ obtained by regressing the continuous variable against the categorical variable.

```
data_association(da, lab=TRUE)
```

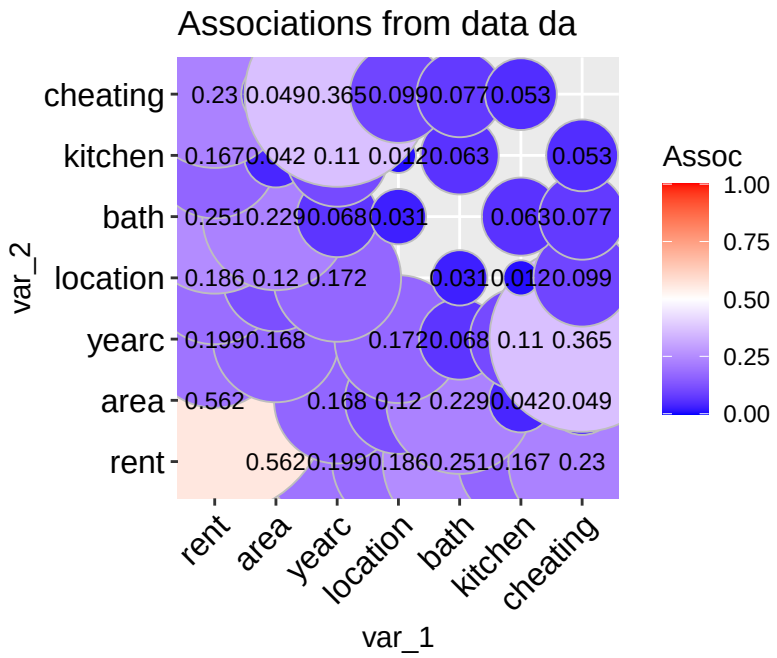
```
 100 % of data are saved,
that is, 3082 observations.
```



A different type of plot can be produce if we use;

```
data_association(da, method="circle", circle.size = 40)
```

```
 100 % of data are saved,
that is, 3082 observations.
```



To get the variables with higher, say, than \$0.4 \$ association values use;

```
Tabcor <- data_cor(da, plot=FALSE)
```

```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_cor(da, plot = FALSE):
```

```
high_val(Tabcor, val=0.4)
```

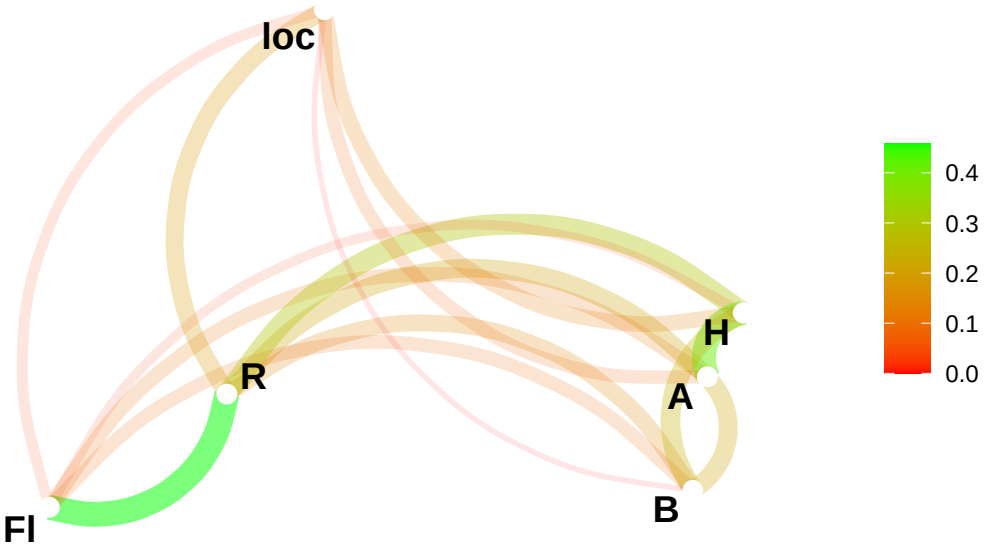
```
     name1  name2  corrs
[1,] "rent" "area" "0.585"
```

A different plot can be acheived using the funtion `network_plot()` of package `corr`;

```
pp=gamlss.prepdata:::data_association(rent[,-c(4,5,8)], plot=F)
```

```
 100 % of data are saved,
that is, 1969 observations.
```

```
library(corrr)
network_plot(pp, min_cor = 0, colors = c("red", "green"), legend = "range")
```



back to table

data_void()

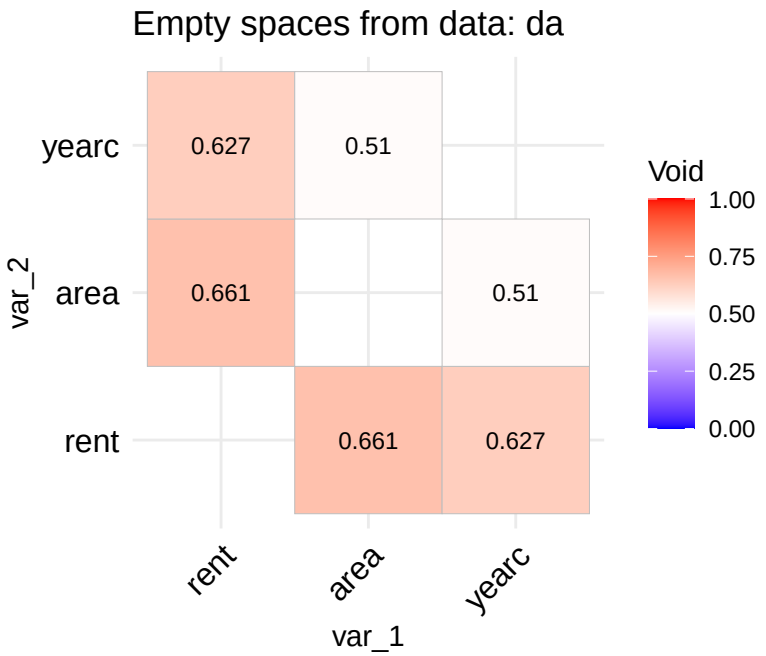
> **Warning**
>
> This function is new. Its theoretical foundation are not proven yet. The function needs testing and therefore it should be used with causion.

The idea behind the functions `void()` and its equivalent `data.frame` version `data_void()` is to be able to identify whether the data in the direction of two continuous variables say x_i and x_j have a lot of empty spaces. The reason is that empty spaces effect prediction since interpolation at empty spaces is dengerous. The function `data_void()` is taking a `data.frame` object and plot the percentage of empty spaces for all pair-wise continuous variables. The function used the `foreach()` function of the package **foreach** to allow parallel processing.

```
registerDoParallel(cores = 9)
data_void(da)
```

```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data
```

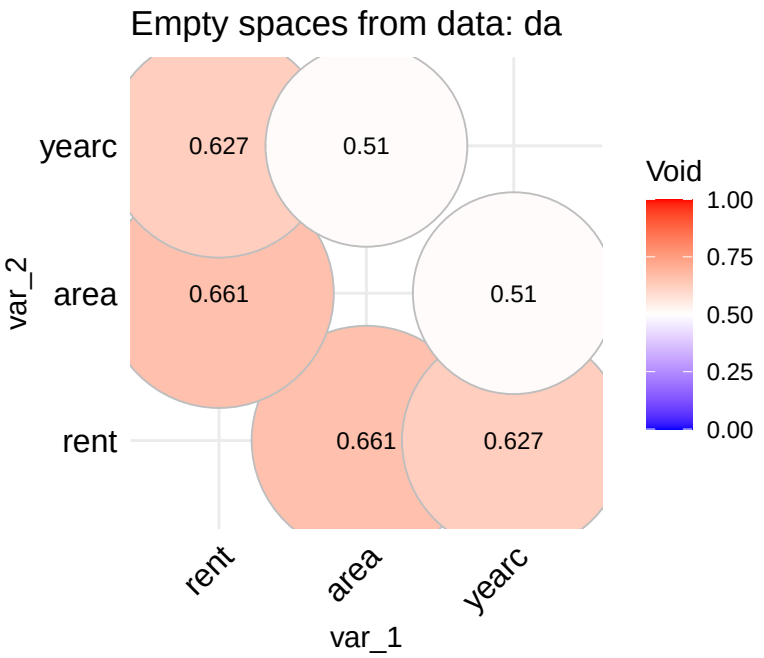
```
Warning in data_void(da):
```



A different type of plot can be produce if we use;

```
data_void(da, method="circle", circle.size = 40)
```

```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_void(da, method = "circle", circle.size = 40):
```




```
stopImplicitCluster()
```

To get the variables with highter than $0.4 $ values use;

```
Tabvoid <- data_void(da, plot=FALSE)
```

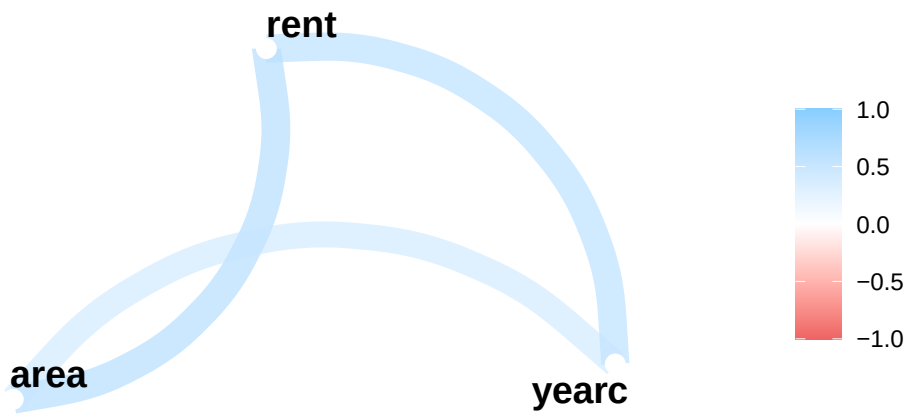
```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_void(da, plot = FALSE):
```

```
high_val(Tabvoid, val=0.4)
```

```
     name1  name2   corrs
[1,] "rent" "area"  "0.661"
[2,] "rent" "yearc" "0.627"
[3,] "area" "yearc" "0.51"
```

To plot their paths use;

```
library(corrr)
network_plot(Tabvoid)
```



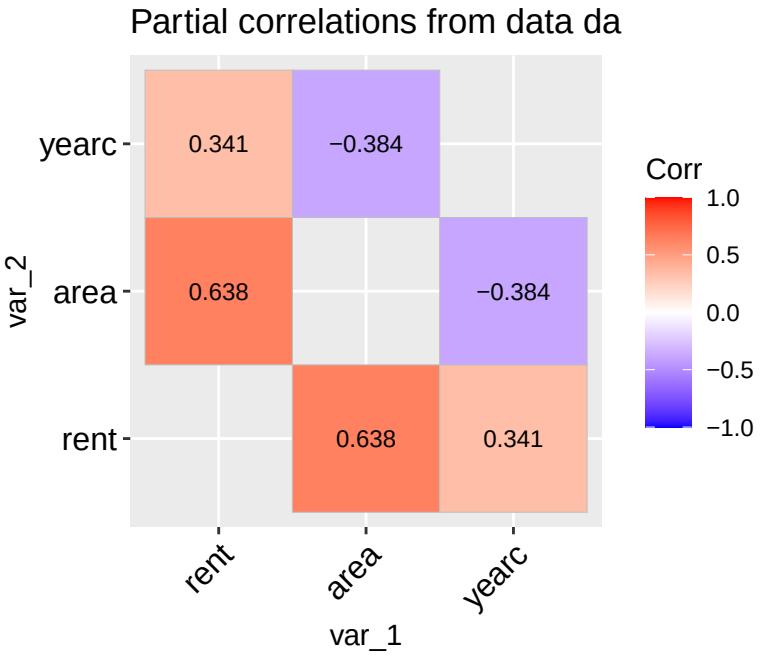
back to table

data_pcor()

The function `data_copr()` is taking a `data.frame` object and plot the partial correlation coefficients of all its continuous variables.

```
data_pcor(da, lab=TRUE)
```

```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_pcor(da, lab = TRUE):
```



To get the variables with highter than $0.4 $ partial correlation values use;

```
Tabpcor <- data_pcor(da, plot=FALSE)
```

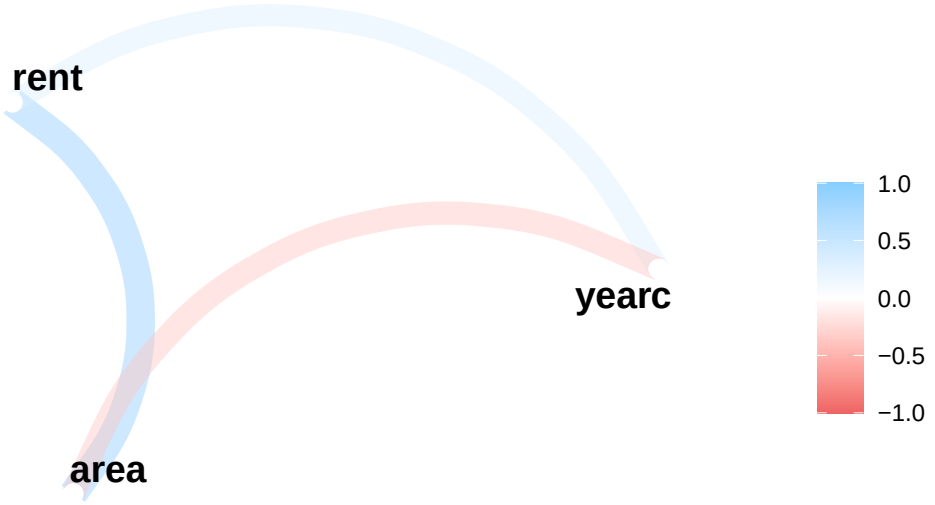
```
 100 % of data are saved,
that is, 3082 observations.
4 factors have been omited from the data

Warning in data_pcor(da, plot = FALSE):
```

```
high_val(Tabpcor, val=0.4)
```

```
     name1  name2  corrs
[1,] "rent" "area" "0.638"
```

For plotting you can use;

```
library(corrr)
network_plot(Tabpcor)
```



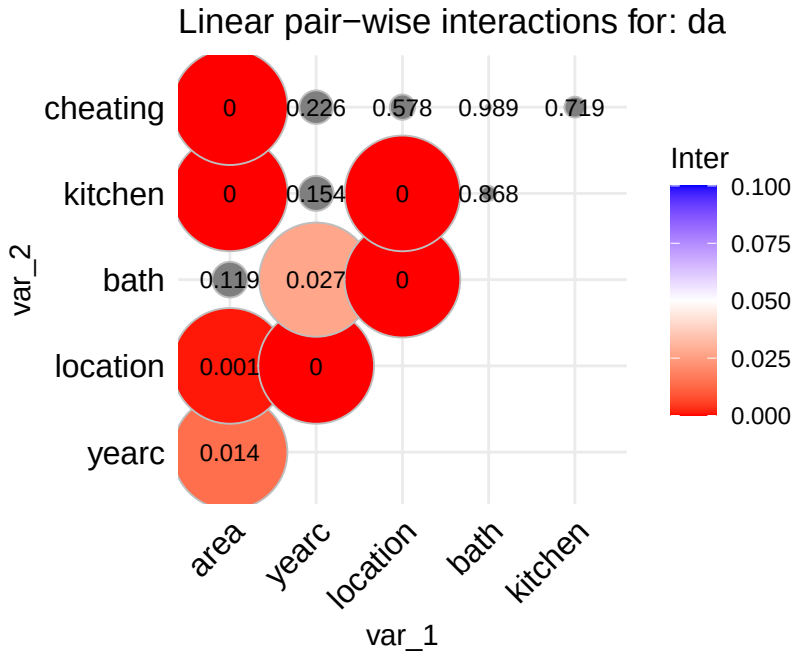
back to table

`data_inter()`

The function `data_inter()` takes a `data.frame`, fits all pair-wise interactions of the explanatory variables against the response (using a normal model) and produce a graph displaying their significance levels. The idea behind this is to identify possible first order interactions at an early stage of the analysis.

```
da |> gamlss.prepdata:::data_inter(response= rent)
```

```
 100 % of data are saved,
that is, 3082 observations.
```




```
tinter <-gamlss.prepdata:::data_inter(da,  response= rent, plot=FALSE)
```

```
 100 % of data are saved,
that is, 3082 observations.
```

```
tinter
```

```
          area yearc location  bath kitchen cheating
area        NA 0.014    0.001 0.119   0.000    0.000
yearc       NA    NA    0.000 0.027   0.154    0.226
location    NA    NA       NA 0.000   0.000    0.578
bath        NA    NA       NA    NA   0.868    0.989
kitchen     NA    NA       NA    NA      NA    0.719
cheating    NA    NA       NA    NA      NA       NA
```

To get the variables with lower than $0.05 $ significant interactions use;

```
Tabinter <- gamlss.prepdata:::data_inter(da, response= rent, plot=FALSE)
```

```
 100 % of data are saved,
that is, 3082 observations.
```

```
low_val(Tabinter)
```

```
     name1      name2      value
[1,] "area"     "yearc"    "0.014"
[2,] "area"     "location" "0.001"
[3,] "yearc"    "location" "0"
[4,] "yearc"    "bath"     "0.027"
[5,] "location" "bath"     "0"
[6,] "area"     "kitchen"  "0"
[7,] "location" "kitchen"  "0"
[8,] "area"     "cheating" "0"
```

back to table

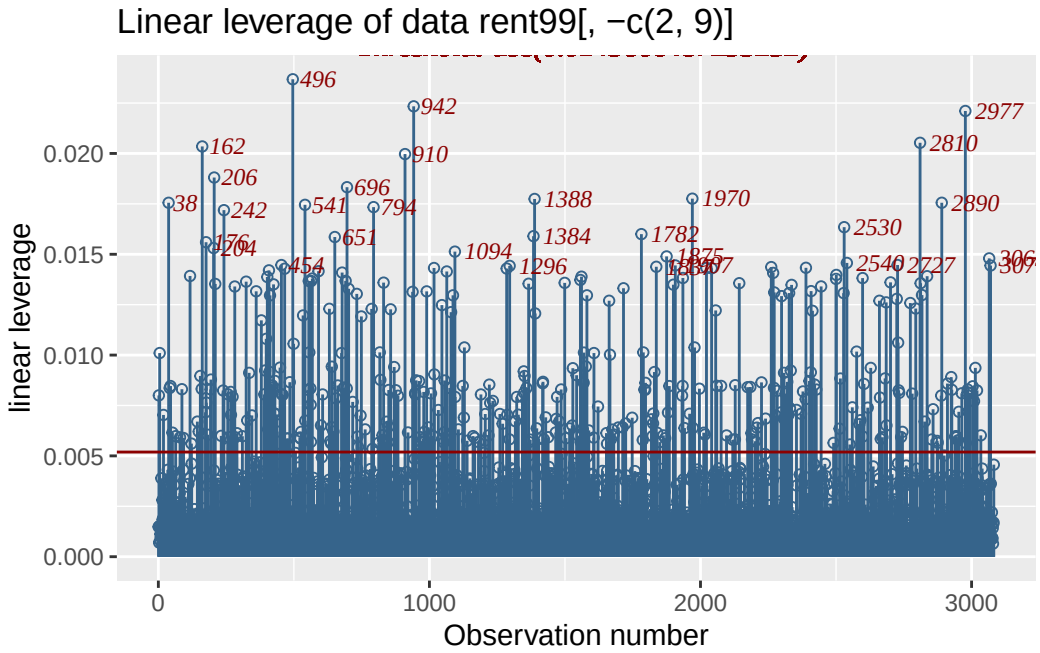
NOT DOCUMENTED IN HELP FOR THIS FUNCTION

data_leverage()

The function `data_leverage()` uses the linear model methodology to identify posible unusual observations within the explanatory variables as a group (not indivisually). It fit a linear (normal) model with response, the response of the data, and all explanatory variables in the data, as x's. It then calculates the leverage points and plots them. A leverage is a number between zero and 1. Large leverage correspond to extremes in the x's.

```
rent99[, -c(2,9)] |> data_leverage( response=rent)
```

```
 100 % of data are saved,
that is, 3082 observations.
```



Note

The horizontal line is the plot is at point $2 \times (r/n)$, which is the threshold suggested in the literature; values beyond this point could be identified as extremes. It looks that the point $2 \times (r/n)$ is too low (at least for our data). Instead the plot identify only observation which are in the one per cent upper quantile. That is a quantile value of the leverage at value 0.99.

Section show how the information of the function `data_leverage()` can be combined with the information given by `data_outliers()` in order to confirm hight outliers in the data.

Important

What happen if multiple response are in the data?

back to table

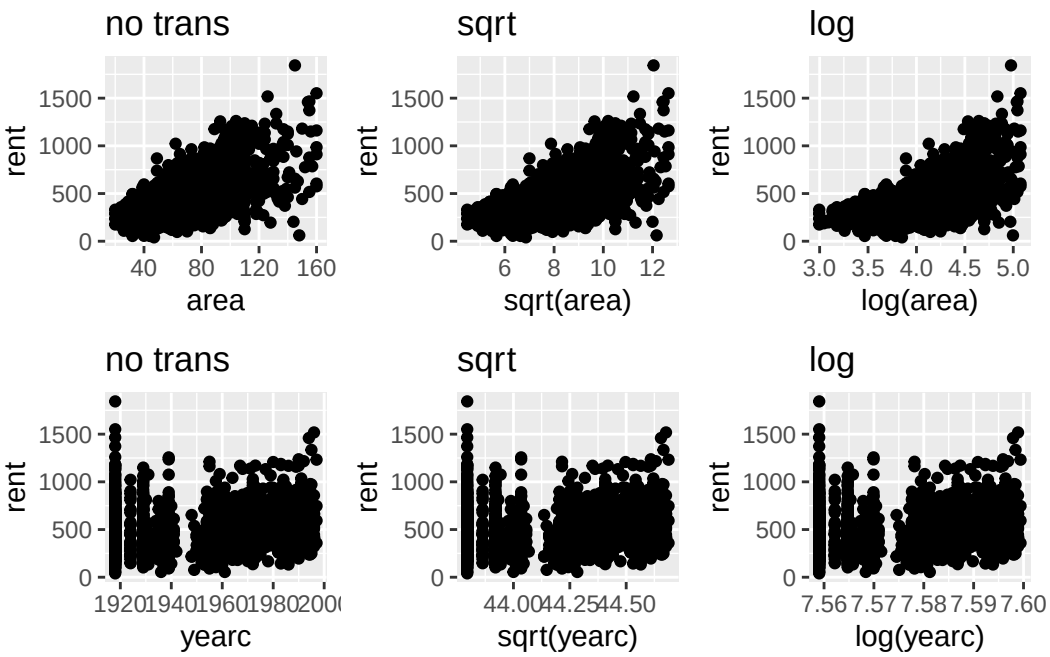
`data_Ptrans_plot()`

The essence of this plot is to visually decide whether certain values of λ in a power transformation are appropriate or not. Section describes the power transformation, $T = X^{\lambda}$ for $\lambda > 0$, in more details. For each continuous explanatory variable X, the function show the response against X, $(\lambda = 1)$, the response against the square root of X, $(\lambda = 1/2)$ and the response

against the log of X ($\lambda \to 0$). The user then can decide whether any of those transformation are appropriate.

```
gamlss.prepdata:::data_Ptrans_plot(da, rent)
```

```
 100 % of data are saved,
that is, 3082 observations.
```



It look that no transformation is needed for the continuous explanatory variables `area` of `yearc`.

back to table

Feature functions

The features functions are functions to help with the continuous explanatory x-variables; We divide them here in four sections; Functions for

- `outliers`;
- `scaling`;
- `transformations` and
- function approproate for `time series`

The available functions are;

Table 3: The outliers, scalling, transformation and time series functions

Functions	Usage
y_outliers()	identify possible outliers in a continuous variable (see below)
data_outliers()	identify possible outliers for all continuous x-variable (see also data_zscores in Section)
data_leverage()	this is a repear of the function from Section
data_scale()	scaling all continuous x-variables either to zero mean and one s.d. or to the range [0,1]
xy_Ptrans()	looking for appropriate power transformation for response against one of the x-variable
data_Ptrans()	looking for appropriate power transformation for response against all x-variable
data_Ptrans_plot()	plotting all x's agains the response using identiti ty square root ans log transormations

Outliers

Outliers in the response variable, within a distributional regression framework like GAMLSS, are better handled by selecting an appropriate distribution for the response. For example, an outlier under the assumption of a normal distribution for the response may no longer be considered an outlier if a distribution from the t-family is assumed. The function `data_response()`, discussed in Section provides guidelines on appropriate actions. This section focuses on outliers in the explanatory variables.

Outliers in the continuous explanatory variables can potentially affect curve fitting, whether using linear or smooth non-parametric terms. In such cases, removing outliers from the x-variables can make the model more robust. Outliers are observations that deviate significantly from the rest of the data, but the concept depends on the dimension being considered. An single observation value might be an outlier within an explanatory variable. A pair observation might be an outlier examining within a pair-wise relationships. In a one-dimensional search, we identify extreme observations in individual continuous variables, based on how far they lie from the majority of the data. In a two-dimensional search, we look for outliers in pairwise plots of continuous variables, examining how observations deviate from typical relationships between variable pairs. Leverage points are useful to identify outliers when we look for outliers in r dimension when r is the number of explanatory variables.

> **Note**
>
> At this preliminary stage of the analysis, where no model has yet been fitted, it is difficult to identify outliers among factors. This is because outliers in factors typically do not appear unusual when examining individual variables. Their outlier status usually becomes apparent only when considered in combination with other variables or factors. Identifying such outliers is a task better suited for the modelling stage of the analysis.

The function `y_outliers()` in Section uses two methodologies to identify outliers; The default uses simple algorithm to identify potential one-dimensional outliers in continuous explanatory variables. It is designed specifically for continuous variables.

- For variables defined on the full real line, (i.e., from $-\infty$ to $+\infty$, the algorithm fits a parametric distribution—by default, the four-parameter `SHASHo` distribution—and computes z-scores (quantile residuals) to assess outlier status.
- For variables defined on the positive real line (i.e., from 0 to $+\infty$), an additional transformation step is included. These variables are often highly right-skewed, and extreme values in the right tail can dominate any outlier detection process. To address this, the function attempts to find a power transformation $T = X^{\lambda}$ that minimizes the Jarque-Bera test statistic—a measure of deviation from normality. To find the optimal power parameters λ the function `y_outliers()` uses the internal function `x_Ptrans()`. After this transformation, the `SHASHo` distribution is fitted, and z-scores are computed. The Jarque-Bera test statistic is always non-negative; the further it is from zero, the stronger the evidence against normality. The goal is to reduce skewness and kurtosis in the data before assessing outliers.

Outliers are identify for observations with very low or hight z-scores. For example, if a z-scores is greater in absolute value to a specified value say 3. The methodology is appropriate for identifying outliers in one dimension. Pair-wise scatter plots of all continuous variables in the data can be use to identify outliers in two dimensions. To identify outliers in p dimensions the function `data_leverage()`, see Section , can be used.

> **Note**
>
> Several of the graphical functions described in Section are good for identify visualy outliers. In fact we recommend that the function `data_outliers()` should be used in combination with functions like `data_plot()` or `data_zscores()`.

`y_outliers()`

The function `y_outliers()` identify outliers in one X variable. There are two methodologies to do that;

- `zscores`: A distribution is fitted to the data (usually `SHASHo`) then the residuals (z-scores) are taken and observations with large residuals are identified (see also the above comment).
- `quantile`: This is the classical methodology and it is based on sample quantiles. An observation is identified if it far from the median. How far usually is taken as three time the semi-inter-quantiles range. The option `value` is set to 4 as a default in the fanction.

Using the zscores;

```
y_outliers(da$rent)
```

```
named integer(0)
```

or using the `quantile`

```
y_outliers(da$rent, type="quantile", value=4)
```

```
 [1]  321  350  354  379  395  404  420  421  424  426  429  431  433  434  438
[16]  439  619  657  675  838  851  856 1015 1016 1576 1877 2005 2070 2271 2551
[31] 2598 2621 2635 2749 2755 2857 2889 2900 2901 2966 2982 3015 3059 3078 3082
```

Note that the `y_outliers()` is used by `data_outliers()` to identify outliers in all continuous variables in the data.

back to table

`data_outliers()`

The function `data_outliers()` uses the *z-scores* technique described above in order to detect outliers in the continuous variables in the data. It fits a `SHASHo` distribution to the continuous variable in the data and uses the z-scores (quantile residuals) to identify (one dimension) outliers for those continuous variables.

```
data_outliers(da)
```

```
    rent     area    yearc location     bath  kitchen cheating
    2723      132       68        3        2        2        2
```

```
$rent
named integer(0)

$area
named integer(0)

$yearc
named integer(0)
```

As it happen no individual variable outliers were highlighted in the `rent` data using the z-scores methodology. In general we recommend the function `data_outliers()` to be used in combination of the graphical functions `data_plot()` and `data_zscores()`

back to table

`data_leverage()` (repeat)

The function `data_leverage()`, fisrt describe in Section , can be used to detect extremes in the continuous variables in the data by fiting a linear model with all the continuous variables in the data and then useing the higher leverage points to identify (multi-dimension) outliers for the observations for all continuous variables. By default, it identifies one percent of the observations with high leverage in the data. Here we use the function as a way to identify the obsrvation (not plottoing).

```
data_leverage(da, response=rent, plot=FALSE)
```

```
 100 % of data are saved,
that is, 3082 observations.

 [1]   38  162  176  204  206  242  454  496  541  651  696  794  910  942 1094
[16] 1296 1384 1388 1782 1837 1875 1907 1970 2530 2540 2727 2810 2890 2977 3065
[31] 3070
```

> **Note**
>
> The function`data_leverage()` always identifies the top one percent of observations with the highest leverage, which may or may not be outliers. These high-leverage points can have a strong influence on model estimates. To better assess their impact, the list of high-leverage observations returned by `data_leverage()` should be compared with the list of outliers identified by the `data_outliers()` function. Observations that appear in both lists are particularly noteworthy, as they may be influential outliers that warrant

> further investigation.

Next we are trying the proceess of identifing if data with high leverage are also coincide with outliers identified using the function `data_outliers()` function. The function `intersect()` will flash out the intersection of the two set of observations. First use `data_outliers()`

```
ul <- unlist(data_outliers(da))
```

```
    rent     area    yearc location     bath  kitchen cheating
    2723      132       68        3        2        2        2
```

then `data_leverage()`;

```
ll<-data_leverage(da, response=rent, plot=FALSE)
```

```
 100 % of data are saved,
that is, 3082 observations.
```

and finally we inspect the intersection of the two sets using `intersect()`;

```
intersect(ll,ul)
```

```
integer(0)
```

Here we have zero outliers but in general we would expect to find common observations for further checking.

back to table

Scaling

Scaling is another form of transformation for the continuous explanatory variables in the data which brings the explanatory variables in the same scale. It is a form of **standardization**. Standardization means bringing all continuous explanatory variables to similar range of values. For some machine learning techniques, i.e., principal component regression or neural networks standardization is mandatory in other like LASSO is recommended. There are two types of standardisation

i) *scaling to normality* and

ii) *scaling to a range from zero to one.*

Scaling to normality it means that the variable should have a mean zero and a standard deviation one. Note that, this is equivalent of fitting a Normal distribution to the variables and then taking the z-scores (residuals) of the fit as the transformed variable. The problem with this type of scaling is that skewness and kurtosis in the standardised data persist. One could go further and fit a four parameter distribution instead of a normal where the two extra parameters could account for skewness and kurtosis. The function `data_scale()` described in Section it gives this option with te argument `family` in which can used to specify a different distribution.

`data_scale()`

The function `data_scale()` perform scaling to all continuous variables an a data set. Note that factors in the data set are left untouched because when fitted within a model they will be transform to dummy variable which take values 0 or 1 (a kind of standardization). The response is left also untouched because it is assumed that an appropriate distribution will be fitted. The response variable has to be declared using the argument `response.` First we use `data_scale()` to scale to normality.

```
head(data_scale(da, response=rent))
```

```
    area    yearc location     bath  kitchen cheating
     132       68        3        2        2        2
```

```
      rent       area      yearc location bath kitchen cheating
1 109.9487  0.7009962  0.7035881        2    0       0        0
2 243.2820 -0.4796117  0.7484206        2    0       0        1
3 261.6410 -0.2266243  0.6587556        1    0       0        1
4 106.4103 -0.3531180 -1.2242096        2    0       0        0
5 133.3846  0.1950214 -1.7173671        2    0       0        1
6 339.0256 -0.5639408  1.6450707        2    0       0        1
```

As we mention before scaling to normality is equivalent of fitting a Normal variable first. The problem with scaling to normality is that if the variables are highly skew or kurtotic scaling them to normality does correct for skewness or kurtosis. Using a parametric distribution with four parameters some of which are skewness and kurtosis parameters it may correct that. Next we standardised using the `SHASHo` distribution;

```
head(data_scale(da, response=rent,  family="SHASHo"))
```

```
    area    yearc location     bath  kitchen cheating
     132       68        3        2        2        2
```

```
      rent       area      yearc location bath kitchen cheating
1 109.9487  0.7390434  0.6323846        2    0       0        0
2 243.2820 -0.3940193  0.6795499        2    0       0        1
3 261.6410 -0.1175129  0.5864434        1    0       0        1
4 106.4103 -0.2526941 -1.0598183        2    0       0        0
5 133.3846  0.2949483 -1.6281459        2    0       0        1
6 339.0256 -0.4916986  2.0610405        2    0       0        1
```

The second form od standardisation is to transform all continuous x-variables to a range zero to one. Here is how this is done;

```
head(data_scale(da, response=rent,  scale.to="0to1"))
```

```
    area    yearc location     bath  kitchen cheating
     132       68        3        2        2        2
```

```
      rent       area yearc location bath kitchen cheating
1 109.9487 0.04285714     0        2    0       0        0
2 243.2820 0.05714286     0        2    0       0        1
3 261.6410 0.07142857     0        1    0       0        1
4 106.4103 0.07142857     0        2    0       0        0
5 133.3846 0.07142857     0        2    0       0        1
6 339.0256 0.07142857     0        2    0       0        1
```

back to table

Transformations

In the context of outliers, the focus on how far a single observation deviates from the rest of the data. However, when considering **transformations**, the goal shifts toward a better capturing of the relationship between a continuous predictor and the response variable. Specifically, we aim to model **peaks and troughs** in the relationship between a single x-variable and the response better. To achieve this, it is sometimes sufficient to stretch the x-axis so that the predictor variable is more evenly distributed within its its range. A common approach for this is to use a power transformation of the form $T = X^{\lambda}$. This transoformation can reduce the curvature in the response and make the curve fitting easier. Importantly, the power transformation smoothly transitions into a logarithmic transformation as $\lambda \to 0$, since:

$\frac{X^\lambda - 1}{\lambda} \rightarrow \log(X)$ as $\lambda \rightarrow 0$. Thus, in the limit as $\lambda \rightarrow 0$ is the power transformation is the log transformation $T = \log(X)$.

> **Note**
>
> The power transformation is a subclass of the shifted power transformation $T = (\alpha + X)^\lambda$ which is not consider here

The aim of the power transformation is to make the model fitting process more robust and reliable and maximizing the information extracted from the data. There are three different functions in `gamlss.prepdata` dealling with power transformations, `xy_Ptrans()`, `data_Ptrans()` and `data_Ptrans_plot()`

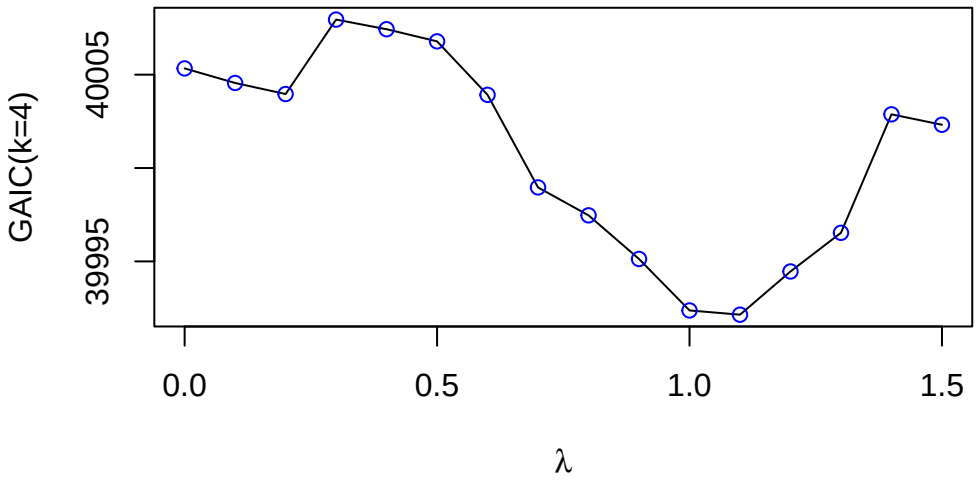
> **Note**
>
> It's important to distinguish between the concepts of *transformations* (the subject of this section) and of *feature extraction*, a term often used in machine learning. We refer to transformations as functions applied to a individual explanatory variables while we use feature extraction when multiple explanatory variables are involved. In both cases, the goal is to enhance the model capabilities.

`xy_Ptrans()`

```
pp<-gamlss.prepdata:::xy_Ptrans(da$area, da$rent)
pp
```

```
[1] 1.06212
```

```
gamlss.prepdata:::xy_Ptrans(da$area, da$rent, prof=TRUE, k=4)
```



data_Ptrans()

```
gamlss.prepdata:::data_Ptrans(da, response=rent)
```

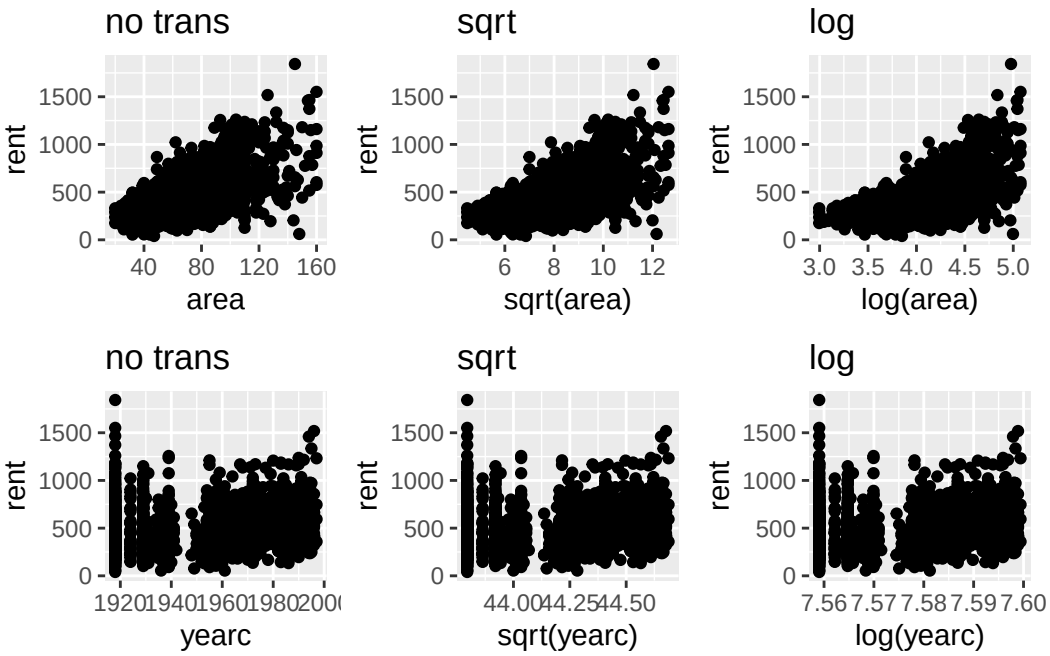
```
 100 % of data are saved,
that is, 3082 observations.

$area
[1] 1.06212

$yearc
[1] 1.499942
```

back to table

data_Ptrans_plot() (repeat)

```
gamlss.prepdata:::data_Ptrans_plot(da, rent)
```

```
 100 % of data are saved,
that is, 3082 observations.
```



back to table

Time series

Time series data sets consist of observations collected at consistent time intervals—e.g., daily, weekly, or monthly. These data require special treatment because observations that are closer together in time tend to be more similar, violating the usual assumption of independence between observations. A similar issue arises in spatial data sets, where observations that are geographically closer often exhibit spatial correlation, again violating independence.

Many data sets include spatial or temporal features —such as longitude and latitude or timestamps (e.g., 12/10/2012)— those variables are can be used for for modeling directly, but also for interpreting or visualization of the results. If temporal or spatial information exists in the data, it is essential to ensure these features are correctly read and interpreted in `R`. For instance, dates in `R` can be handled using the `as.Date()` function. To understand its functionality and options associated with the function please consult the help documentation via `help("as.Date")`. Below, we provide two simple functions of how to work with date-time variables.

`time_dt2dhour()`

Suppose the variable `dt` contains both a date and a time. We may want to extract and separate this into two components: one containing the date, and another capturing the hour of the day.

```
dt <- c("01/01/2011 00:00", "01/01/2011 01:00", "01/01/2011 02:00", "01/01/2011 03:00", "01/0
```

The function to do this separation can lok like;

```
time_dt2dhour <- function(datetime, format=NULL)
  {
          X <- t(as.data.frame(strsplit(datetime,' ')))
rownames(X) <- NULL
colnames(X) <- c("date", "time")
       hour <- as.numeric(sub(":",".",X[,2]))
      date <- as.Date(X[,1],format=format)
      data.frame(date, hour)
  }
```

```
P <- time_dt2dhour(dt, "%d/%m/%Y")
P
```

```
        date hour
1 2011-01-01    0
2 2011-01-01    1
3 2011-01-01    2
4 2011-01-01    3
5 2011-01-01    4
6 2011-01-01    5
```

time_2num()

The second example contains time given in its common format, `c("12:35", "10:50")` and we would like t change it to numeric which we can use in the model.

```
t <- c("12:35", "10:50")
```

The function to change this to numeric;

```
time_2num <- function(time, pattern=":")
  {
  t <- gsub(pattern, ".", time)
   as.numeric(t)
  }
time_2num(t)
```

```
[1] 12.35 10.50
```

For user who deal with a lot of time series try the function `POSXct()` and also the function `select()` from the `dplur` package.

Data Partition

Introdunction to data partition

Partitioning the data is essential for comparing different models and checking for overfitting. Figure Figure 1 illustrates the different ways the data can be split. A dataset can be split multiple times or just once, depending on the size of the dataset. For large datasets, it is common to split the data into a training set and a test set, and, if necessary, a validation set. Observations in the training set are referred to as in-bag`, while observations in the test or validation set are termed`out-of-bag‘.

The training set is used for fitting the model, the test set is used for prediction and comparison between models, and the validation set is used for tuning the model's `hyperparameters`. In additive smoothing regression models, for instance, the hyperparameters refer to the `smoothing` parameters in the model. Data partitioning allows for performance evaluation, model assumption checks, and detection of overfitting.

If the fitted model performs reasonably well on both the training and test sets, one can be confident that overfitting has been avoided. Overfitting occurs when a model fails to generalize well to new data, typically because it is too closely fitted to the sample data. In contrast, underfitting refers to a poor model that fails to capture the essential patterns in the data, resulting in a model that is distant from both the sample and the population it is intended to represent.

Various goodness-of-fit measures can be used to check and compare different distributional regression models. One of the most popular methods is the Generalized Akaike Information Criterion (GAIC). To calculate the GAIC, only the in-bag data are required, meaning there is no need for data partitioning. However, calculating the GAIC requires an accurate measure of the degrees of freedom used to fit the model. In mathematical (stochastic) models, the degrees of freedom are straightforward to define as they correspond to the number of estimated parameters. In contrast, for algorithmic models, particularly over-parameterized ones like neural networks, estimating degrees of freedom is more challenging. In mathematical models, the degrees of freedom serve as a measure of the model's complexity. More complex models are typically closer to the training data, which may reduce their ability to generalize well to new data.

In addition to GAIC, other commonly used goodness-of-fit measures for regression models include the deviance, R^2, and mean squared error (MSE), among others. However, these

measures are prone to overfitting when evaluated on the in-bag dataset rather than on the out-of-bag data. Furthermore, R^2 may not generalize well to non-normal error distributions, and MSE (and its variations) is often inappropriate for distributional regression models. This is because MSE focuses primarily on the location parameters of the response distribution, while other aspects of the response distribution are often of greater interest to practitioners.

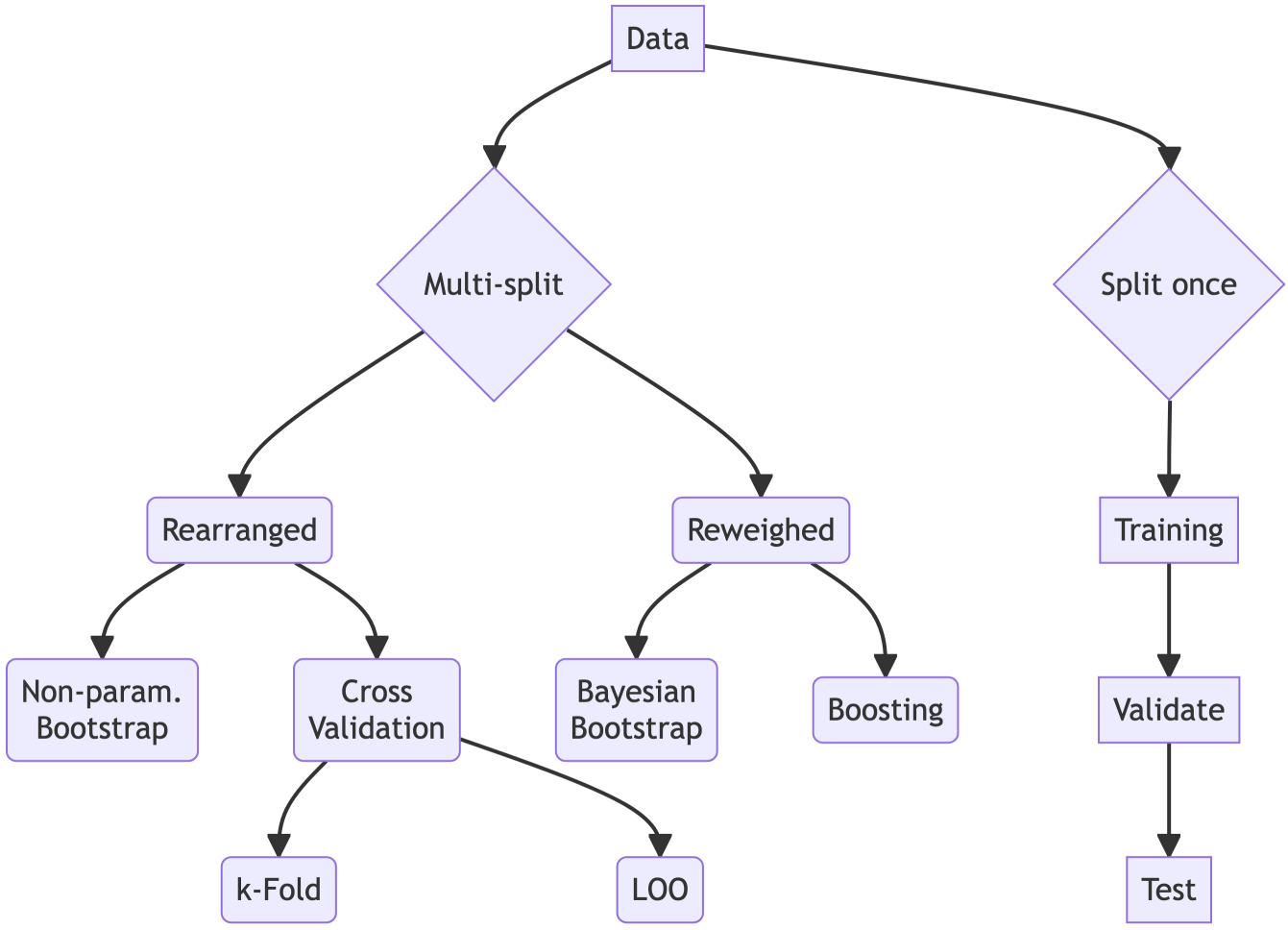


Figure 1: Different ways of splitting data to get in order to get more information"

Partitioning the data into training, test, and validation sets should be done randomly. A potential risk when splitting the data only once arises when the investigation focuses on the tails of the response distribution. In such cases, the few extreme observations may end up in only one of the partitions, leading to potential issues when evaluating the model. Repeatedly reusing parts of the data, such as through bootstrapping or cross-validation, can help mitigate this risk. Multi-splitting the data (shown on the left side of Figure Figure 1) enables the evaluation of appropriate goodness-of-fit measures using out-of-bag data. There are two con-

ceptual ways to multi-split a dataset: the first involves rearranging the data, and the second involves reweighting the observations. [As we will see later, the distinction between these two approaches is not always as clear-cut.] Rearranging the data is typically accomplished by creating appropriate indices that select different parts of the data. This leads to methods such as **bootstrapping** or **cross-validation**. On the other hand, reweighting the observations during the fitting process is a technique employed in the **Bayesian Bootstrap** and **boosting**.

In classical bootstrapping, the data are re-sampled randomly with replacement B times, and the model is refitted the same number of times. While computationally expensive, this process provides additional insights into the model, particularly regarding the variability of its parameters. For distributional regression models, both classical and Bayesian bootstrapping offer extra information about all the parameters in the model—namely, the distributional parameters, fitted coefficients, hyperparameters, and random effects. The Bayesian bootstrap re-weights the data as samples from a multinomial distribution and fits the re-weighted models B times. Like the classical bootstrap, it provides valuable information about the variability of the model's parameters. In boosting, each weak learner (i.e., each simple fitted model) is reweighted using the residuals from the previous fits. The results of all these learners are then aggregated into the final model. Boosting can be seen as a method of iteratively improving model accuracy by focusing on the previously mispredicted observations.

In K-fold cross-validation, the data are split into K sub-samples. The model is then fitted k times, each time using $k-1$ sub-samples for training and the remaining one sub-sample as the test set. By the end of the process, all n observations will have been used as test data exactly once. The main advantage of K-fold cross-validation is that it provides reliable test data for model evaluation with the additional cost of fitting K models. Leave-One-Out Cross-Validation (LOO) is a special case of k-fold cross-validation, where the model is trained on $n-1$ observations and tested on the remaining single observation, iterating this process n times. Consequently, n different models are refitted. While computationally intensive, LOO ensures that each data point is used for testing, providing an unbiased evaluation of model performance. Both bootstrapping and K-fold cross-validation, when appropriate performance metrics are used, can help in several ways: i) by facilitating model comparison on the test data, ii) by avoiding overfitting, and iii) by assisting in the selection of hyperparameters.

Notice that in distributional regression models, there is no inherent need to physically partition the original data into subgroups when performing a single or multiple partitions. The same results can be achieved by indexing the original data.frame or by using prior weights during the model fitting process. For example, in K-fold cross-validation, we need K dummy vectors, i_k, where \$k = 1, ..., K. These dummy vectors take the value of 1 for the k-th set of observations and 0 for the rest. The dummy vectors i_k can be used in two ways:

1. To select the observations from the `data.frame` for fitting.
2. As prior weights when fitting the model.

Let `da` represent the `data.frame` used to fit the model, and i the i_k dummy vector. For each of the K fits, the model fitting process would use the command $data = da[, i]$ to select the appropriate data, and `newdata=da[,i==0]` for evaluating the out-of-bag measure. Alternatively, one can use `weights=i` during the model fitting and `newdata=da[,i==0]` for evaluating any out-of-bag measures. Note that the appropriate measures for comparing models include predictive deviance (PD) and continuous rank probability scores (CRPS). The function data_Kfold() in the package generates the correct dummy vectors as a matrix of dimensions n ×B, which can be used for the cross-validation process.

For the classical bootstrap, we need B vectors. Unlike cross-validation (CV), where the vectors for indexing and prior weights are identical, bootstrapping requires different vectors. The function `data_boot_index()` creates the appropriate vectors for indexing, while `data_boot_weights()` generates the vectors for prior weighting. These vectors should differ whether used as indices to select the relevant columns from the data matrix `da` or as prior weights, which indicate how many times each observation is selected. Typically, a vector like $(1, 0, 2, ..., 3, 2)$ means that observation 1 is picked once, observation 2 is excluded from the fit, observation 3 is selected twice, and so on. For Bayesian bootstrap, the B vectors of length n represent weights that sum to n. An out-of-bag observation, in this case, is one with zero weight. This distinction makes the split between rearranged and reweighed in Figure 1 somewhat artificial, as both methods can be interpreted as refitting approaches using prior weights.

Table 4: The data partition functions

Functions	Usage
`data_part()`	Creates a single or multiple (CV) partitions by introducing a factor with different levels
`data_part_list()`	Creates a single or multiple (CV) partitions with output a list of `data.frames`
`data_boot_index()`	Creates two lists. The in-bag,`IB`, indices for fitting and the out-of-bag, `OOB`, indices for prediction
`data_boot_weights()`	Create a $n \times B$ matrix with columns weights for bootstrap fits
`data_Kfold_index()`	Creates a $n \times K$ matrix of variables to be used for cross validation data indexing
`data_Kfold_weights()`	Creates a $n \times K$ matrix of dummy variables to be used for cross validation weighted fits
`data_cut()`	This is not a partition funtion but randomly select a specified porpotion of the data

Here are the data partition functions.

data_part()

The function `data_part()` it does not partitioning the data as such but adds a factor called `partition` to the data indicating the partition. By default the data are partitioned into two sets the training ,`train`, and the test data, `test`.

```
dap <- gamlss.prepdata:::data_part(da)
```

```
data partition into two sets
```

```
head(dap)
```

```
      rent area yearc location bath kitchen cheating partition
1 109.9487   26  1918        2    0       0        0     train
2 243.2820   28  1918        2    0       0        1      test
3 261.6410   30  1918        1    0       0        1     train
4 106.4103   30  1918        2    0       0        0      test
5 133.3846   30  1918        2    0       0        1      test
6 339.0256   30  1918        2    0       0        1     train
```

If the user would like to split the data in two separate `data.frame`'s she/he can use;

```
daTrain <- subset(dap, partition=="train")
daTest <- subset(dap, partition=="test")
dim(daTrain)
```

```
[1] 1846    8
```

```
dim(daTest)
```

```
[1] 1236    8
```

Note, that use of the option `partition` has the following behaviour;

i) `partition=2L` (the default) the factor has two levels `train`, and `test`.

ii) `partition=3L` the factor has three levels `train`, `val` (for validation) and `test`.

iii) `partition > 4L` say `K` then the levels are `1`, `2`...`K`. The factor then can be used to identify K-fold cross validation, see also the function `data_Kfols_CV()`.

back to table

data_part_list()

The function `data_part_list()` creates a list of `dataframes` which can be used either for single partition or for cross validation. The argument `partition` allows the splitting of the data to up to 20 subsets. The default is a list of 2 with elements `training` and `test` (single partition).

Here is a single partition in `train` and `test` data sets.

```
allda <-  data_part_list(rent)
length(allda)
```

```
[1] 2
```

```
dim(allda[["train"]]) # training data
```

```
[1] 1200    9
```

```
dim(allda[["test"]]) # test data
```

```
[1] 769   9
```

Here is a multiple partition for cross validation.

```
allda <-  data_part_list(rent, partition=10)
```

```
10-fold data partition
```

```
length(allda)
```

```
[1] 10
```

```
names(allda)
```

```
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

back to table

data_boot_index()

The function `data_boot_index()` create two lists. The first called `IB` (for in-bag) and the second `OOB`, (out-of-bag). Each ellement of the list `IB` can be use to select a bootstrap sample from the original `data.frame` while each element of `OOB` can be use to select the out of sample data points. Note that each bootstap sample approximately contains 2/3 of the original data so we expact on avarage to 1/3 to be in `OOB` sample.

```
DD <- data_boot_index(rent, B=10)
```

The class, length and names of the created lists;

```
class(DD)
```

```
[1] "list"
```

```
length(DD)
```

```
[1] 2
```

```
names(DD)
```

```
[1] "IB"  "OOB"
```

the first obsevations of the

```
head(DD$IB[[1]]) # in bag
```

```
[1] 1 2 4 6 8 8
```

```
head(DD$OOB[[1]]) # out-of-bag
```

```
[1]  3  5  7 15 21 22
```

back to table

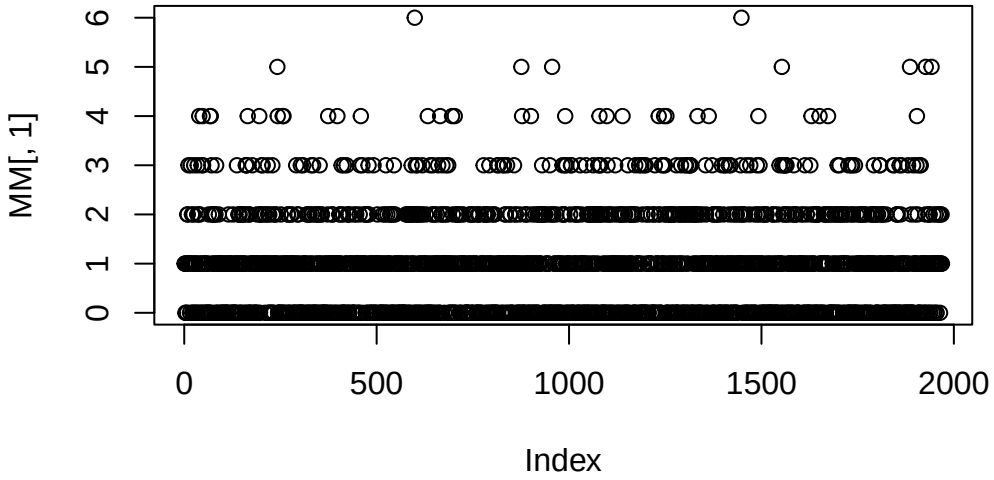
data_boot_weights()

The function `data_boot_weights()` create a $n \times B$. matrix with columns possible weights for bootstap fits. Note that each bootstap sample approximately contains .666 of the original obsbation so we in general we expect 0.333 of the data to have zero weights.

```
MM <- data_boot_weights(rent, B=10)
```

The `M` is a matrix which columns can be used as weights in bootstrap fits; Here is a plot of the first column

```
plot(MM[,1])
```



and here is the first 6 rows os the matrix;

```
head(MM)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    2    1    0    1    3    1    3     1
[2,]    1    1    0    2    2    1    2    1    1     2
[3,]    0    0    2    1    2    1    2    1    1     0
[4,]    1    0    1    0    3    0    1    1    0     2
[5,]    0    2    3    1    1    1    2    1    2     1
[6,]    1    0    0    1    0    1    4    0    1     0
```

back to table

data_Kfold_index()

The function `data_Kfold()` creates a matrix which columns can be use for cross validation either as indices to select data for fitting or as prior weights.

```
PP <- data_Kfold_index(rent, K=10)
head(PP)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    0    1    1    1    1     1
[2,]    2    2    0    2    2    2    2    2    2     2
[3,]    3    3    3    3    3    3    3    3    0     3
[4,]    4    4    4    4    4    0    4    4    4     4
[5,]    5    5    5    5    0    5    5    5    5     5
[6,]    6    6    6    6    6    6    6    0    6     6
```

back to table

data_Kfold_weights()

The function `data_Kfold()` creates a matrix which columns can be use for cross validation either as indices to select data for fitting or as prior weights.

```
PP <- data_Kfold_weights(rent, K=10)
head(PP)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    0    1    1    1    1     1
[2,]    1    1    0    1    1    1    1    1    1     1
[3,]    1    1    1    1    1    1    1    1    0     1
[4,]    1    1    1    1    1    0    1    1    1     1
[5,]    1    1    1    1    0    1    1    1    1     1
[6,]    1    1    1    1    1    1    1    0    1     1
```

back to table

`data_cut()`

The function `data_cut()` is not included in the partition Section for its data partitioning properties. It is designed to select a random subset of the data specifically for plotting purposes. This is especially useful when working with large datasets, where plotting routines—such as those from the `ggplot2` package—can become very slow.

The `data_cut()` function can either: • Automatically reduce the data size based on the total number of observations, or • Subset the data according to a user-specified percentage.

Here is an example of the function assuming that that the user requires 50 of the data;

```
da1 <- data_cut(rent99, percentage=.5)
```

```
 50 % of data are saved,
that is, 1541 observations.
```

```
dim(rent99)
```

```
[1] 3082    9
```

```
dim(da1)
```

```
[1] 1541    9
```

The function `data_cut()` is used extensively in many plotting routines within the `gamlss.ggplots` package. When the percentage option is not specified, a default rule is applied to determine the proportion of the data used for plotting. This approach balances performance and visualization quality when working with large datasets.

Let n denote the total number of observations. Then:

- if $n \leq 50,000$: All data are used for plotting.
- if $50,000 < n \leq 100,000$: 50% of the data are used.
- If $100,000 < n \leq 1,000,000$: 20% of the data are used.
- If $n > 1,000,000$: 10% of the data are used.

This default behaviour ensures that plotting remains efficient even with very large datasets.

back to table

Distribution Families

> **Note**
>
> All functions listed below belong to the `gamlss.ggplots` package, not the `gamlss.prepdata` package. However, since they can be useful during the pre-modeling stage, they are presented here. The later version of the `gamlss.ggplots` can be found in https://github.com/gamlss-dev/gamlss.ggplots

In the following function, no fitted model is required—only values for the parameters need to be specified.

`family_pdf()`

The function `family_pdf()` plots individual probability density functions (PDFs) from distributions in the `gamlss.family` package. Although it typically requires the family argument, it defaults to the Normal distribution (NO) if none is provided.

```
family_pdf(from=-5,to=5, mu=0, sigma=c(.5,1,2))
```

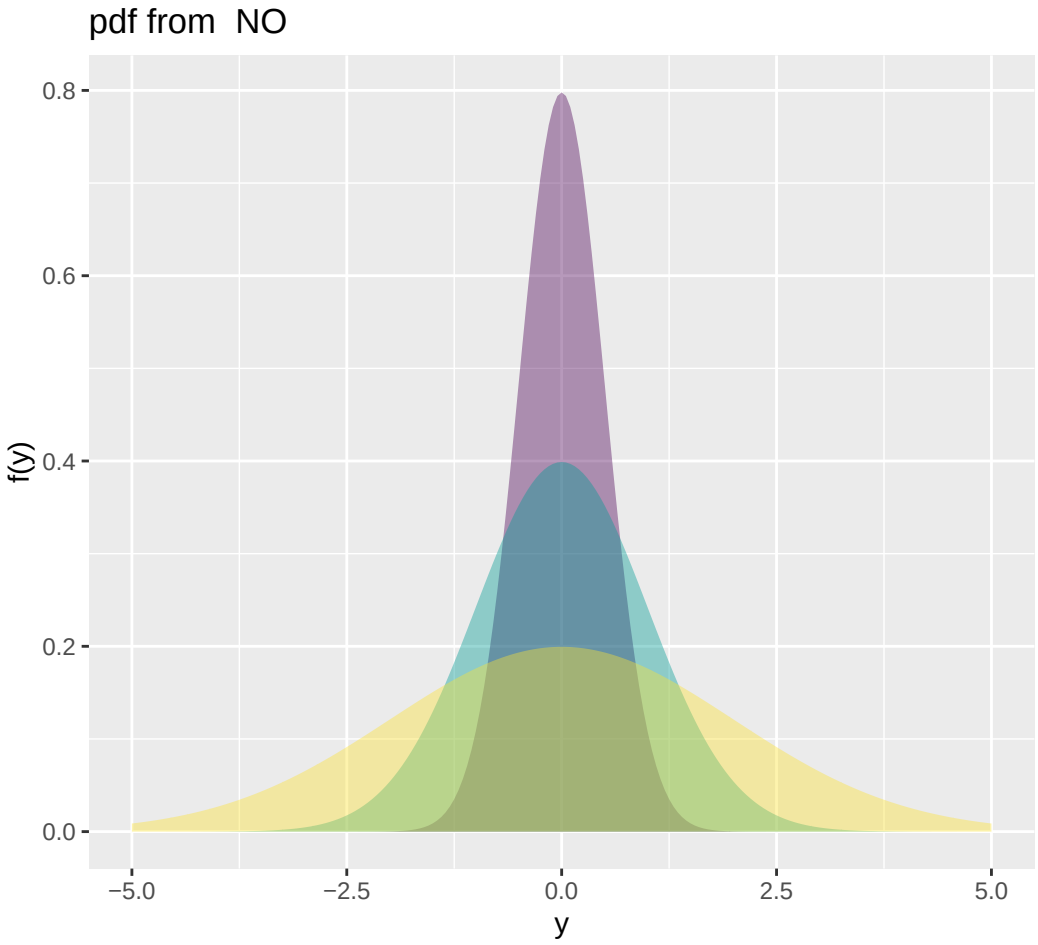


Figure 2: Continuous response example: plotting the pdf of a normal random variable.

The following demonstrates how discrete distributions are displayed. We begin with a distribution that can take infinitely many count values—the Negative Binomial distribution;

```
family_pdf(NBI, to=15, mu=1, sigma=c(.5,1,2), alpha=.9, size.seqment = 3)
```

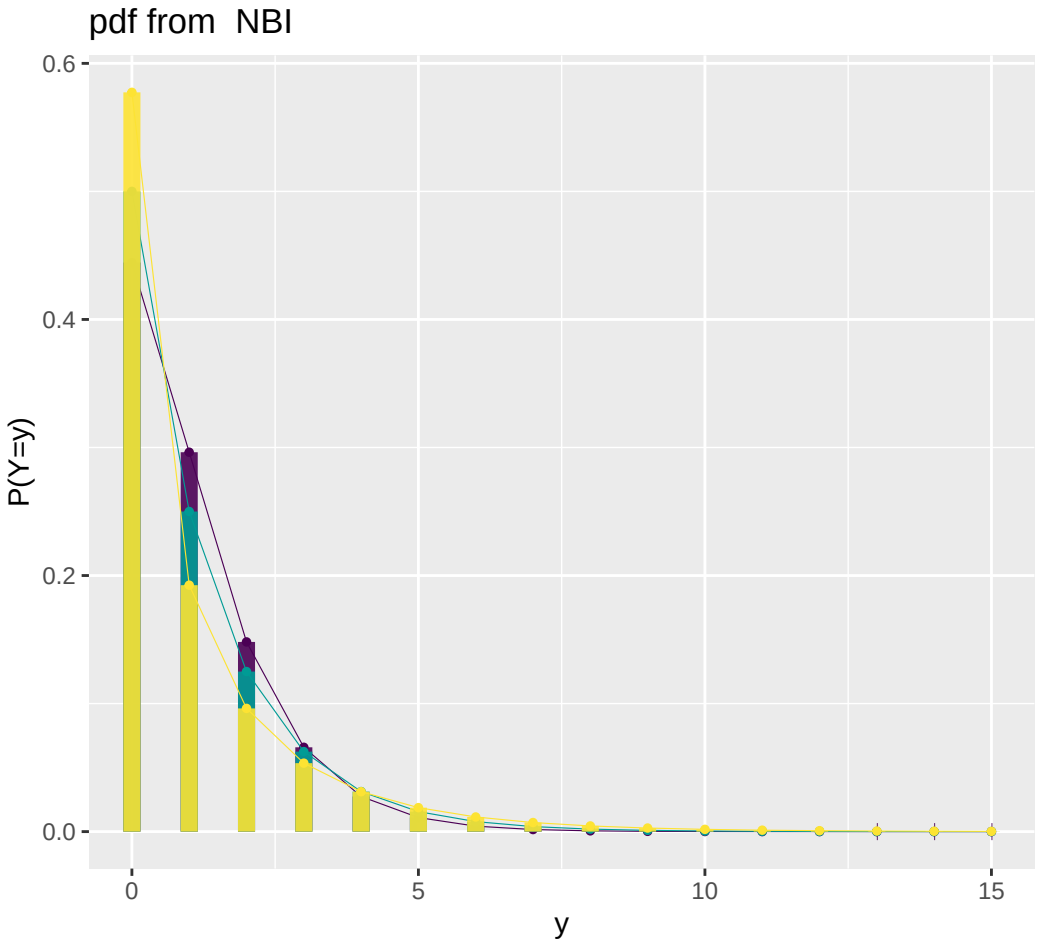


Figure 3: Count response example: plotting the pdf of a beta binomial.

Here we use a discrete distribution with a finite range of possible values: the Beta-Binomial distribution.

```
family_pdf(BB, to=15, mu=.5, sigma=c(.5,1,2),  alpha=.9, , size.seqment = 3)
```

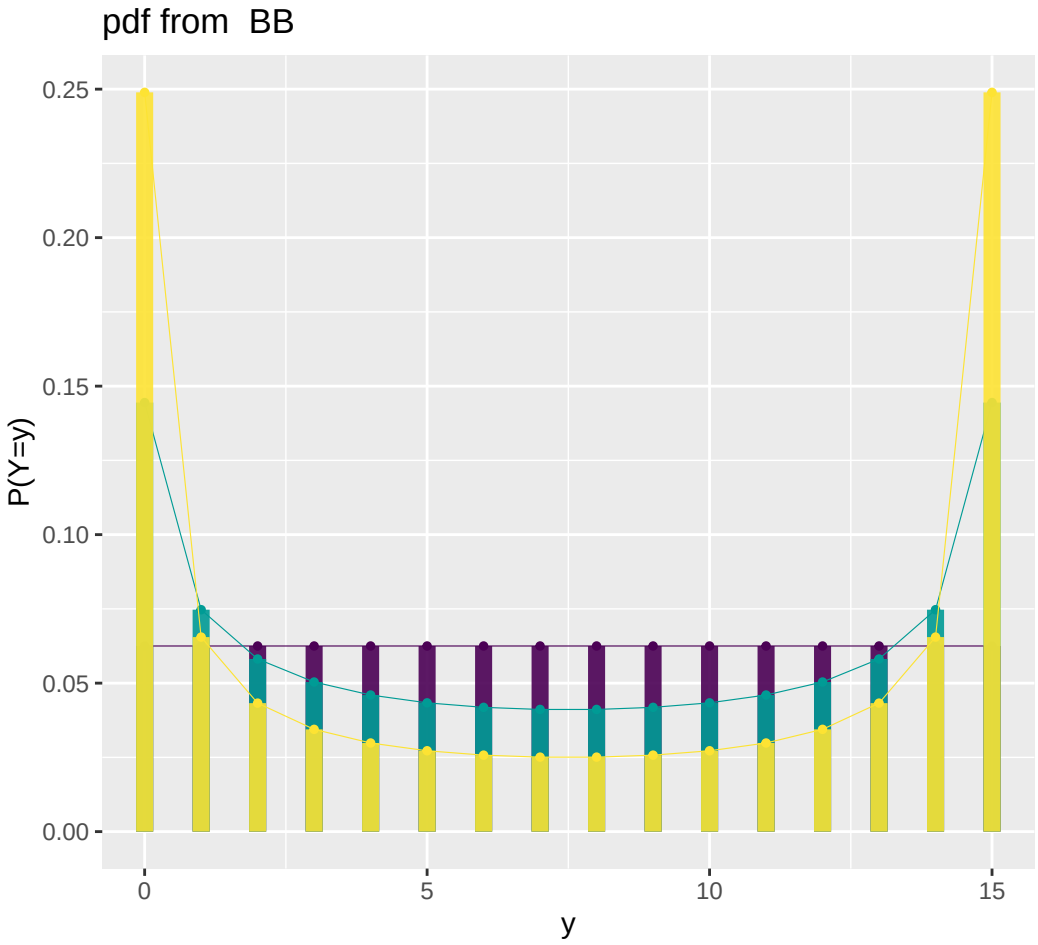


Figure 4: Beta binomial response example: plotting the pdf of a beta binomial.

`family_cdf`

The function `family_cdf()`

The function `family_cdf()` plots cumulative distribution functions (CDFs) from the **gamlss.family** distributions. The primary argument required is the family specifying the distribution to be used.

```
family_cdf(NBI, to=15, mu=1, sigma=c(.5,1,2), alpha=.9, size.seqment = 3)
```

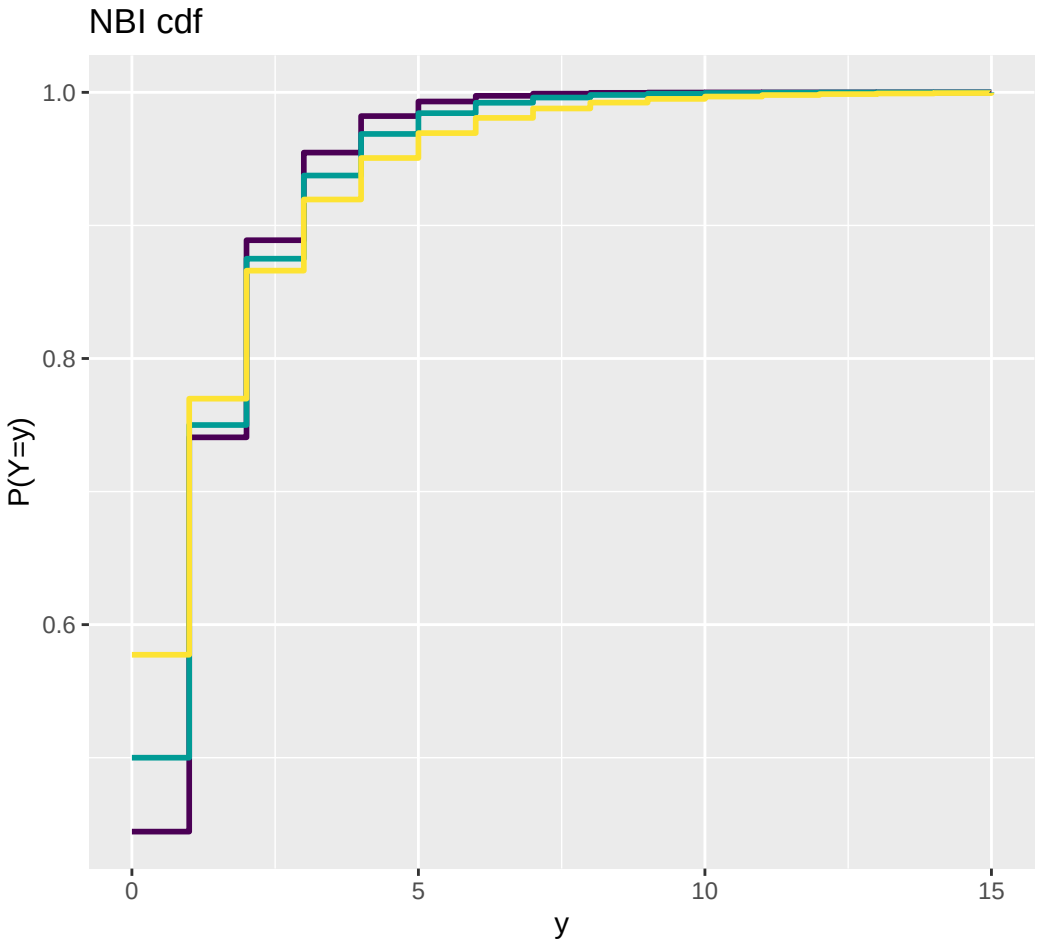


Figure 5: Count response example: plotting the cdf of a negative binomial.

The CDf of the negative binomial;

```
family_cdf(BB, to=15, mu=.5, sigma=c(.5,1,2),  alpha=.9, , size.seqment = 3)
```

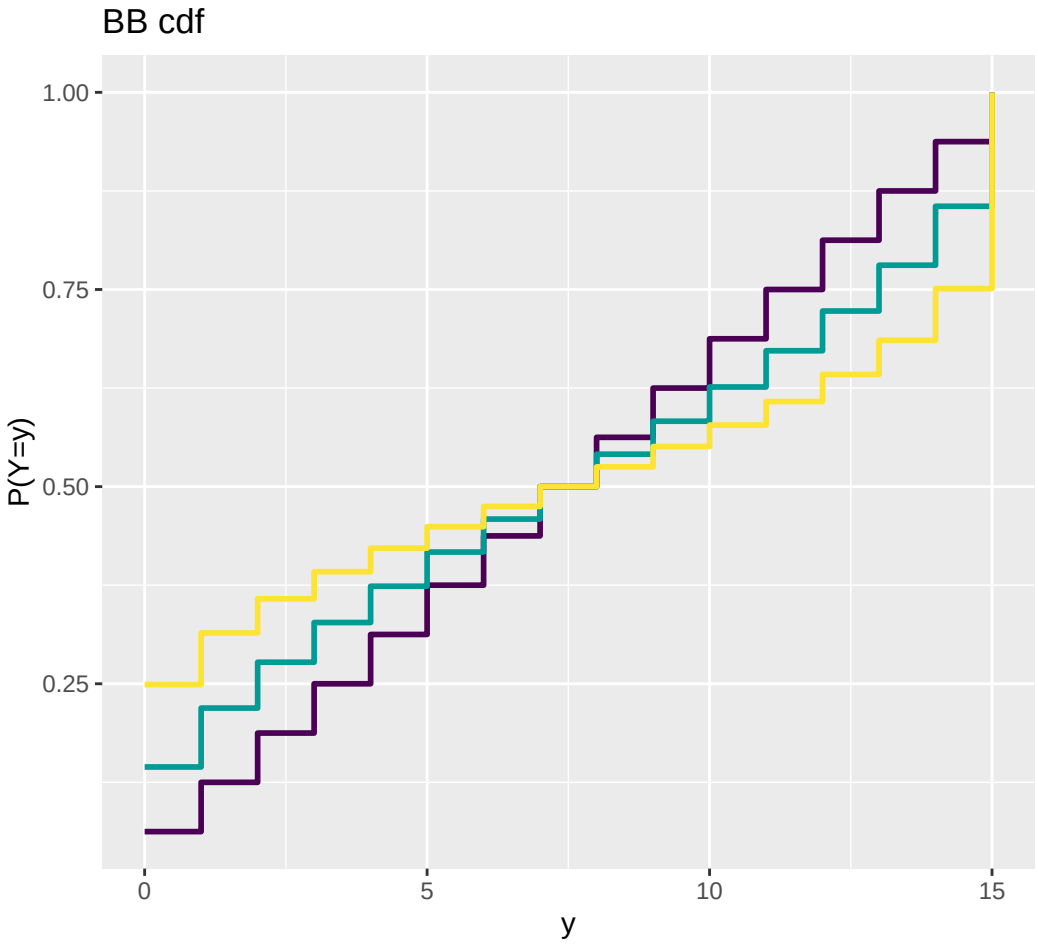


Figure 6: Count response example: plotting the cdf of a beta binomial.

`family_cor()`

The function `family_cor()` offers a basic method for examining the inter-correlation among the parameters of any distribution from the **gamlss.family**. It performs the following steps:

- Generates 10,000 random values from the specified distribution.
- Fits the same distribution to the generated data.
- Extracts and plots the correlation coefficients of the distributional parameters.

These correlation coefficients are derived from the variance-covariance matrix of the fitted model.

> **Warning**
>
> This method provides only a rough indication of how the parameters are correlated at specific values of the distribution's parameters. The correlation structure may vary significantly at different points in the parameter space, as the distribution can behave quite differently depending on those values.

```
#source("~/Dropbox/github/gamlss-ggplots/R/family_cor.R")
gamlss.ggplots:::family_cor("BCTo", mu=1, sigma=0.11, nu=1, tau=5, no.sim=10000)
```

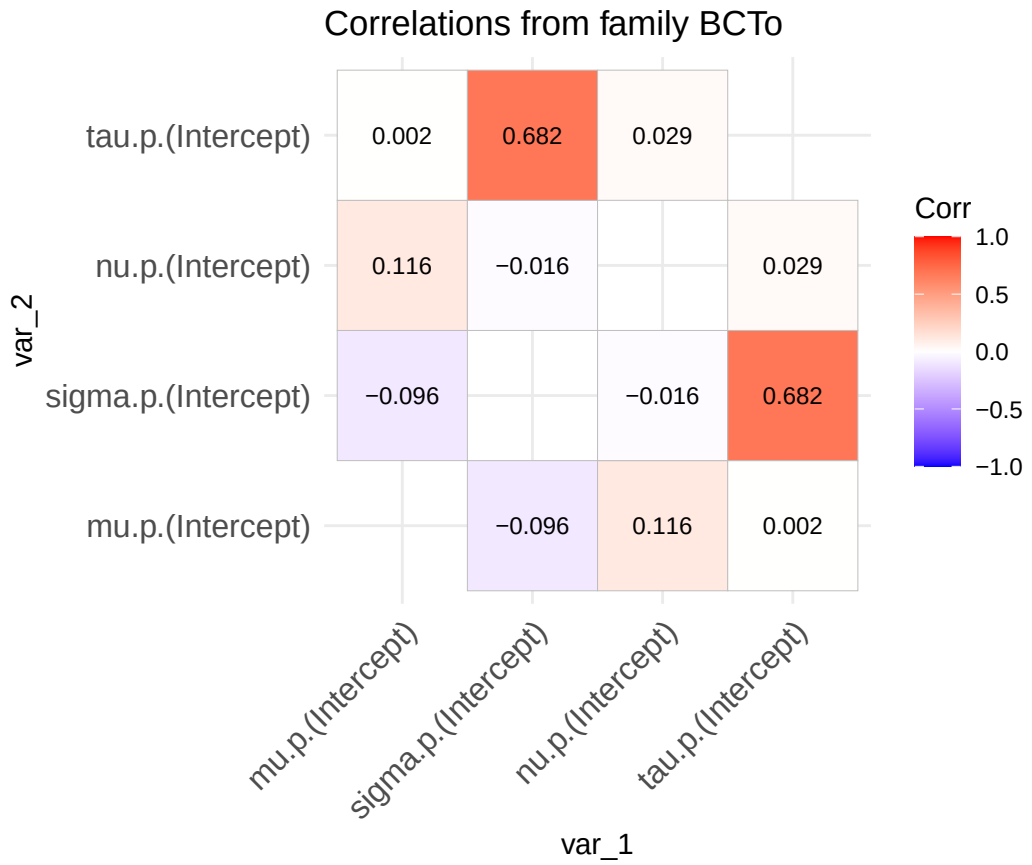


Figure 7: Family correlation of a BCTo distribution at specified values of the parameters.

Rigby, R. A., and D. M. Stasinopoulos. 2005. "Generalized Additive Models for Location, Scale and Shape (with Discussion)." *Applied Statistics* 54: 507–54.

Rigby, R. A., D. M. Stasinopoulos, G. Z. Heller, and F. De Bastiani. 2019. *Distributions for Modeling Location, Scale, and Shape: Using GAMLSS in r*. Boca Raton: Chapman & Hall/CRC.

Stasinopoulos, D. M., R. A. Rigby, G. Z. Heller, V. Voudouris, and F. De Bastiani. 2017. *Flexible Regression and Smoothing: Using GAMLSS in r.* Boca Raton: Chapman & Hall/CRC.

Stasinopoulos, M. D., T. Kneib, N. Klein, A. Mayr, and G. Z Heller. 2024. *Generalized Additive Models for Location, Scale and Shape: A Distributional Regression Approach, with Applications.* Vol. 56. Cambridge University Press.