

# Package ‘reservr’

December 9, 2022

**Title** Fit Distributions and Neural Networks to Censored and Truncated Data

**Version** 0.0.1

**Description** Define distribution families and fit them to interval-censored and interval-truncated data, where the truncation bounds may depend on the individual observation. The defined distributions feature density, probability, sampling and fitting methods as well as efficient implementations of the log-density  $\log f(x)$  and log-probability  $\log P(x_0 \leq X \leq x_1)$  for use in 'TensorFlow' neural networks via the 'tensorflow' package. Allows training parametric neural networks on interval-censored and interval-truncated data with flexible parameterization. Applications include Claims Development in Non-Life Insurance, e.g. modelling reporting delay distributions from incomplete data, see Bücher, Rosenstock (2022) <[doi:10.1007/s13385-022-00314-4](https://doi.org/10.1007/s13385-022-00314-4)>.

**License** GPL

**BugReports** <https://github.com/AshesITR/reservr/issues>

**Depends** R (>= 3.5)

**Imports** assertthat (>= 0.2.1), generics, glue (>= 1.3.1), keras (>= 2.2.5.0), matrixStats, nloptr, numDeriv, purrr (>= 0.3.3), R6 (>= 2.4.1), Rcpp, RcppParallel, rlang (>= 0.4.5), stats, utils

**Suggests** covr, callr, colorspace, data.table, dplyr (>= 0.8.4), evmix, fitdistrplus (>= 1.0.14), flextable (>= 0.5.8), formattable (>= 0.2.0.1), furr (>= 0.1.0), ggplot2 (>= 3.2.1), ggribes (>= 0.5.2), knitr (>= 1.28), logKDE (>= 0.3.2), officer (>= 0.3.7), patchwork (>= 1.0.0), reticulate, rmarkdown (>= 2.1), rstudioapi, tensorflow (>= 2.0.0), testthat (>= 2.1.0), tidyr (>= 1.0.2), tibble

**LinkingTo** BH, Rcpp, RcppArmadillo, RcppParallel

**VignetteBuilder** knitr

**Encoding** UTF-8

**RoxygenNote** 7.2.2

**SystemRequirements** GNU make

**Collate** 'ReppExports.R' 'fit\_util.R' 'distribution\_class.R' 'zzz.R'  
 'interval.R' 'aaa.R' 'blended\_transition.R'  
 'callback\_adaptive\_lr.R' 'callback\_debug\_dist\_gradients.R'  
 'check\_lengths.R' 'compiler.R' 'dist\_bdegp.R' 'dist\_beta.R'  
 'dist\_binomial.R' 'fit\_blended.R' 'dist\_blended.R'  
 'dist\_dirac.R' 'dist\_discrete.R' 'dist\_empirical.R'  
 'fit\_erng\_mixture.R' 'dist\_erngmix.R' 'dist\_exponential.R'  
 'dist\_gamma.R' 'gpd.R' 'dist\_genpareto.R' 'dist\_lognormal.R'  
 'fit\_mixture.R' 'dist\_mixture.R' 'dist\_negbinomial.R'  
 'dist\_normal.R' 'pareto.R' 'dist\_pareto.R' 'dist\_poisson.R'  
 'dist\_translate.R' 'dist\_trunc.R' 'dist\_uniform.R'  
 'dist\_weibull.R' 'distribution\_generics.R'  
 'distribution\_methods.R' 'flatten\_params.R' 'integrate.R'  
 'plot\_distributions.R' 'prob\_report.R' 'reservr-package.R'  
 'softmax.R' 'tf\_compile.R' 'tf\_compile\_loss.R' 'tf\_constants.R'  
 'tf\_fit.R' 'tf\_initialise.R' 'tf\_util.R'  
 'trunc\_erngmix\_init.R' 'trunc\_obs.R' 'truncate\_claims.R'  
 'weighted\_stats.R'

**URL** <https://ashesitr.github.io/reservr/>

**NeedsCompilation** yes

**Author** Alexander Rosenstock [aut, cre, cph]

**Maintainer** Alexander Rosenstock <alexander.rosenstock@web.de>

**Repository** CRAN

**Date/Publication** 2022-12-09 14:40:02 UTC

**R topics documented:**

as_params . . . . .	3
blended_transition . . . . .	4
callback_adaptive_lr . . . . .	6
callback_debug_dist_gradients . . . . .	8
Distribution . . . . .	9
dist_bdegp . . . . .	23
dist_beta . . . . .	24
dist_binomial . . . . .	25
dist_blended . . . . .	26
dist_dirac . . . . .	27
dist_discrete . . . . .	28
dist_empirical . . . . .	29
dist_erngmix . . . . .	31
dist_exponential . . . . .	32
dist_gamma . . . . .	33
dist_genpareto . . . . .	34
dist_lognormal . . . . .	35

dist_mixture . . . . .	36
dist_negbinomial . . . . .	37
dist_normal . . . . .	38
dist_pareto . . . . .	39
dist_poisson . . . . .	40
dist_translate . . . . .	41
dist_trunc . . . . .	42
dist_uniform . . . . .	43
dist_weibull . . . . .	44
fit.reservr_keras_model . . . . .	45
fit_blended . . . . .	48
fit_dist . . . . .	49
fit_dist_start.MixtureDistribution . . . . .	51
fit_erlang_mixture . . . . .	52
fit_mixture . . . . .	53
flatten_params . . . . .	55
GenPareto . . . . .	56
integrate_gk . . . . .	58
interval . . . . .	59
interval-operations . . . . .	60
is.Distribution . . . . .	62
k_matrix . . . . .	62
Pareto . . . . .	63
plot_distributions . . . . .	64
predict.reservr_keras_model . . . . .	65
prob_report . . . . .	66
quantile.Distribution . . . . .	68
softmax . . . . .	69
tf_compile_model . . . . .	70
tf_initialise_model . . . . .	72
truncate_claims . . . . .	73
trunc_obs . . . . .	74
weighted_moments . . . . .	76
weighted_quantile . . . . .	77
weighted_tabulate . . . . .	78
<b>Index</b>	<b>79</b>

---

as\_params

*Convert TensorFlow tensors to distribution parameters recursively*


---

### Description

Convert TensorFlow tensors to distribution parameters recursively

### Usage

as\_params(x)

**Arguments**

`x` possibly nested list structure of tensorflow.tensors

**Value**

A nested list of vectors suitable as distribution parameters

**Examples**

```
if (interactive() && keras::is_keras_available()) {
  tf_params <- list(
    probs = k_matrix(t(c(0.5, 0.3, 0.2))),
    shapes = k_matrix(t(c(1L, 2L, 3L)), dtype = "int32"),
    scale = keras::k_constant(1.0)
  )
  params <- as_params(tf_params)
  dist <- dist_erlangmix(vector("list", 3L))
  dist$sample(10L, with_params = params)
}
```

---

blended\_transition      *Transition functions for blended distributions*

---

**Description**

Transition functions for blended distributions

**Usage**

```
blended_transition(x, u, eps, .gradient = FALSE, .extend_na = FALSE)
```

```
blended_transition_inv(x, u, eps, .component)
```

**Arguments**

<code>x</code>	Points to evaluate at
<code>u</code>	Sorted vector of blending thresholds, or rowwise sorted matrix of blending thresholds
<code>eps</code>	Corresponding vector or matrix of blending bandwidths. Must be positive and the same dimensions as <code>u</code> , or scalar. No rowwise blending regions ( <code>u - eps</code> , <code>u + eps</code> ) may overlap.
<code>.gradient</code>	Also evaluate the gradient with respect to <code>x</code> ?
<code>.extend_na</code>	Extend out-of range transitions by the last in-range value (i.e. the corresponding <code>u</code> ) or by NA?
<code>.component</code>	Component index (up to <code>length(u) + 1</code> ) to invert.

**Value**

`blended_transition` returns a matrix with `length(x)` rows and `length(u) + 1` columns containing the transformed values for each of the blending components. If `.gradient` is `TRUE`, an attribute `"gradient"` is attached with the same dimensions, containing the derivative of the respective transition component with respect to `x`.

`blended_transition_inv` returns a vector with `length(x)` values containing the inverse of the transformed values for the `.componentth` blending component.

**Examples**

```
library(ggplot2)
xx <- seq(from = 0, to = 20, length.out = 101)
blend_mat <- blended_transition(xx, u = 10, eps = 3, .gradient = TRUE)
ggplot(
  data.frame(
    x = rep(xx, 2L),
    fun = rep(c("p", "q"), each = length(xx)),
    y = as.numeric(blend_mat),
    relevant = c(xx <= 13, xx >= 7)
  ),
  aes(x = x, y = y, color = fun, linetype = relevant)
) %+%
  geom_line() %+%
  theme_bw() %+%
  theme(
    legend.position = "bottom", legend.box = "horizontal"
  ) %+%
  guides(color = guide_legend(direction = "horizontal", title = ""), linetype = guide_none()) %+%
  scale_linetype_manual(values = c("TRUE" = 1, "FALSE" = 3))

ggplot(
  data.frame(
    x = rep(xx, 2L),
    fun = rep(c("p'", "q'"), each = length(xx)),
    y = as.numeric(attr(blend_mat, "gradient")),
    relevant = c(xx <= 13, xx >= 7)
  ),
  aes(x = x, y = y, color = fun, linetype = relevant)
) %+%
  geom_line() %+%
  theme_bw() %+%
  theme(
    legend.position = "bottom", legend.box = "horizontal"
  ) %+%
  guides(color = guide_legend(direction = "horizontal", title = ""), linetype = guide_none()) %+%
  scale_linetype_manual(values = c("TRUE" = 1, "FALSE" = 3))
```

---

callback\_adaptive\_lr *Keras Callback for adaptive learning rate with weight restoration*

---

### Description

Provides a keras callback similar to `keras::callback_reduce_lr_on_plateau()` but which also restores the weights to the best seen so far whenever a learning rate reduction occurs, and with slightly more restrictive improvement detection.

### Usage

```
callback_adaptive_lr(
  monitor = "val_loss",
  factor = 0.1,
  patience = 10L,
  verbose = 0L,
  mode = c("auto", "min", "max"),
  delta_abs = 1e-04,
  delta_rel = 0,
  cooldown = 0L,
  min_lr = 0,
  restore_weights = TRUE
)
```

### Arguments

monitor	quantity to be monitored.
factor	factor by which the learning rate will be reduced. $\text{new\_lr} = \text{old\_lr} * \text{factor}$ .
patience	number of epochs with no significant improvement after which the learning rate will be reduced.
verbose	integer. Set to 1 to receive update messages.
mode	Optimisation mode. "auto" detects the mode from the name of monitor. "min" monitors for decreasing metrics. "max" monitors for increasing metrics.
delta_abs	Minimum absolute metric improvement per epoch. The learning rate will be reduced if the average improvement is less than <code>delta_abs</code> per epoch for <code>patience</code> epochs.
delta_rel	Minimum relative metric improvement per epoch. The learning rate will be reduced if the average improvement is less than $ \text{metric}  * \text{delta\_rel}$ per epoch for <code>patience</code> epochs.
cooldown	number of epochs to wait before resuming normal operation after learning rate has been reduced. The minimum number of epochs between two learning rate reductions is <code>patience + cooldown</code> .
min_lr	lower bound for the learning rate. If a learning rate reduction would lower the learning rate below <code>min_lr</code> , it will be clipped at <code>min_lr</code> instead and no further reductions will be performed.

restore\_weights

Bool. If TRUE, the best weights will be restored at each learning rate reduction. This is very useful if the metric oscillates.

### Details

Note that while `callback_reduce_lr_on_plateau()` automatically logs the learning rate as a metric 'lr', this is currently impossible from R. Thus, if you want to also log the learning rate, you should add `callback_reduce_lr_on_plateau()` with a high `min_lr` to effectively disable the callback but still monitor the learning rate.

### Value

A KerasCallback suitable for passing to `keras::fit()`.

### Examples

```
dist <- dist_exponential()
group <- sample(c(0, 1), size = 100, replace = TRUE)
x <- dist$sample(100, with_params = list(rate = group + 1))
global_fit <- fit(dist, x)

if (keras::is_keras_available()) {
  library(keras)
  l_in <- layer_input(shape = 1L)
  mod <- tf_compile_model(
    inputs = list(l_in),
    intermediate_output = l_in,
    dist = dist,
    optimizer = optimizer_adam(),
    censoring = FALSE,
    truncation = FALSE
  )
  tf_initialise_model(mod, global_fit$params)
  fit_history <- fit(
    mod,
    x = k_constant(group),
    y = as_trunc_obs(x),
    epochs = 20L,
    callbacks = list(
      callback_adaptive_lr("loss", factor = 0.5, patience = 2L, verbose = 1L, min_lr = 1.0e-4),
      callback_reduce_lr_on_plateau("loss", min_lr = 1.0) # to track lr
    )
  )

  plot(fit_history)

  predicted_means <- predict(mod, data = k_constant(c(0, 1)))
}
```

---

`callback_debug_dist_gradients`*Callback to monitor likelihood gradient components*

---

### Description

Provides a keras callback to monitor the individual components of the censored and truncated likelihood. Useful for debugging TensorFlow implementations of Distributions.

### Usage

```
callback_debug_dist_gradients(  
  object,  
  data,  
  obs,  
  keep_grads = FALSE,  
  stop_on_na = TRUE,  
  verbose = TRUE  
)
```

### Arguments

<code>object</code>	A <code>resvr_keras_model</code> created by <code>tf_compile_model()</code> .
<code>data</code>	Input data for the model.
<code>obs</code>	Observations associated to data.
<code>keep_grads</code>	Log actual gradients? (memory hungry!)
<code>stop_on_na</code>	Stop if any likelihood component as NaN in its gradients?
<code>verbose</code>	Print a message if training is halted? The Message will contain information about which likelihood components have NaN in their gradients.

### Value

A `KerasCallback` suitable for passing to `keras::fit()`.

### Examples

```
dist <- dist_exponential()  
group <- sample(c(0, 1), size = 100, replace = TRUE)  
x <- dist$sample(100, with_params = list(rate = group + 1))  
global_fit <- fit(dist, x)  
  
if (keras::is_keras_available()) {  
  library(keras)  
  l_in <- layer_input(shape = 1L)  
  mod <- tf_compile_model(  
    inputs = list(l_in),  
    intermediate_output = l_in,  
  )  
}
```



```

    dist = dist,
    optimizer = optimizer_adam(),
    censoring = FALSE,
    truncation = FALSE
  )
  tf_initialise_model(mod, global_fit$params)
  # TODO update when rstudio/keras#1230 is fixed
  gradient_tracker <- callback_debug_dist_gradients(mod, k_constant(group), x, keep_grads = TRUE)
  fit_history <- fit(
    mod,
    x = k_constant(group),
    y = x,
    epochs = 20L,
    callbacks = list(
      callback_adaptive_lr("loss", factor = 0.5, patience = 2L, verbose = 1L, min_lr = 1.0e-4),
      gradient_tracker,
      callback_reduce_lr_on_plateau("loss", min_lr = 1.0) # to track lr
    )
  )
  gradient_tracker$gradient_logs[[20]]$dens

  plot(fit_history)

  predicted_means <- predict(mod, data = k_constant(c(0, 1)))
}

```

---

 Distribution

*Base class for Distributions*


---

## Description

Represents a modifiable Distribution family

## Active bindings

`default_params` Get or set (non-recursive) default parameters of a Distribution

`param_bounds` Get or set (non-recursive) parameter bounds (box constraints) of a Distribution

## Methods

### Public methods:

- `Distribution$new()`
- `Distribution$sample()`
- `Distribution$density()`
- `Distribution$tf_logdensity()`
- `Distribution$probability()`
- `Distribution$tf_logprobability()`

- `Distribution$quantile()`
- `Distribution$hazard()`
- `Distribution$diff_density()`
- `Distribution$diff_probability()`
- `Distribution$is_in_support()`
- `Distribution$is_discrete_at()`
- `Distribution$tf_is_discrete_at()`
- `Distribution$has_capability()`
- `Distribution$get_type()`
- `Distribution$get_components()`
- `Distribution$is_discrete()`
- `Distribution$is_continuous()`
- `Distribution$require_capability()`
- `Distribution$get_dof()`
- `Distribution$get_placeholders()`
- `Distribution$get_params()`
- `Distribution$tf_make_constants()`
- `Distribution$tf_compile_params()`
- `Distribution$get_param_bounds()`
- `Distribution$get_param_constraints()`
- `Distribution$export_functions()`
- `Distribution$clone()`

**Method** `new()`:

*Usage:*

`Distribution$new(type, caps, params, name, default_params)`

*Arguments:*

`type` Type of distribution. This is a string constant for the default implementation. Distributions with non-constant type must override the `get_type()` function.

`caps` Character vector of capabilities to fuel the default implementations of `has_capability()` and `require_capability()`. Distributions with dynamic capabilities must override the `has_capability()` function.

`params` Initial parameter bounds structure, backing the `param_bounds` active binding (usually a list of intervals).

`name` Name of the Distribution class. Should be CamelCase and end with "Distribution".

`default_params` Initial fixed parameters backing the `default_params` active binding (usually a list of numeric / NULLs).

*Details:* Construct a Distribution instance  
Used internally by the `dist_*` functions.

**Method** `sample()`:

*Usage:*

`Distribution$sample(n, with_params = list())`

*Arguments:*

`n` number of samples to draw.

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `n`. In that case the *i*-th sample will use the *i*-th parameters.

*Details:* Sample from a Distribution

*Returns:* A length `n` vector of i.i.d. random samples from the Distribution with the specified parameters.

*Examples:*

```
dist_exponential(rate = 2.0)$sample(10)
```

**Method** `density()`:*Usage:*

```
Distribution$density(x, log = FALSE, with_params = list())
```

*Arguments:*

`x` Vector of points to evaluate the density at.

`log` Flag. If TRUE, return the log-density instead.

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `length(x)`. In that case, the *i*-th density point will use the *i*-th parameters.

*Details:* Density of a Distribution

*Returns:* A numeric vector of (log-)densities

*Examples:*

```
dist_exponential()$density(c(1.0, 2.0), with_params = list(rate = 2.0))
```

**Method** `tf_logdensity()`:*Usage:*

```
Distribution$tf_logdensity()
```

*Details:* Compile a TensorFlow function for log-density evaluation

*Returns:* A `tf_function` taking arguments `x` and `args` returning the log-density of the Distribution evaluated at `x` with parameters `args`.

**Method** `probability()`:*Usage:*

```
Distribution$probability(
  q,
  lower.tail = TRUE,
  log.p = FALSE,
  with_params = list()
)
```

*Arguments:*

`q` Vector of points to evaluate the probability function at.

`lower.tail` If TRUE, return  $P(X \leq q)$ . Otherwise return  $P(X > q)$ .

`log.p` If TRUE, probabilities are returned as  $\log(p)$ .

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `length(q)`. In that case, the *i*-th probability point will use the *i*-th parameters.

*Details:* Cumulative probability of a Distribution

*Returns:* A numeric vector of (log-)probabilities

*Examples:*

```
dist_exponential()$probability(
  c(1.0, 2.0),
  with_params = list(rate = 2.0)
)
```

**Method** `tf_logprobability()`:

*Usage:*

```
Distribution$tf_logprobability()
```

*Details:* Compile a TensorFlow function for log-probability evaluation

*Returns:* A `tf_function` taking arguments `qmin`, `qmax` and `args` returning the log-probability of the Distribution evaluated over the closed interval `[qmin, qmax]` with parameters `args`.

**Method** `quantile()`:

*Usage:*

```
Distribution$quantile(
  p,
  lower.tail = TRUE,
  log.p = FALSE,
  with_params = list()
)
```

*Arguments:*

`p` Vector of probabilities.

`lower.tail` If TRUE, return  $P(X \leq q)$ . Otherwise return  $P(X > q)$ .

`log.p` If TRUE, probabilities are returned as  $\log(p)$ .

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `length(p)`. In that case, the *i*-th quantile will use the *i*-th parameters.

*Details:* Quantile function of a Distribution

*Returns:* A numeric vector of quantiles

*Examples:*

```
dist_exponential()$quantile(c(0.1, 0.5), with_params = list(rate = 2.0))
```

**Method** `hazard()`:

*Usage:*

```
Distribution$hazard(x, log = FALSE, with_params = list())
```

*Arguments:*

`x` Vector of points.

`log` Flag. If TRUE, return the log-hazard instead.

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `length(x)`. In that case, the *i*-th hazard point will use the *i*-th parameters.

*Details:* Hazard function of a Distribution

*Returns:* A numeric vector of (log-)hazards

*Examples:*

```
dist_exponential(rate = 2.0)$hazard(c(1.0, 2.0))
```

**Method** `diff_density()`:

*Usage:*

```
Distribution$diff_density(x, log = FALSE, with_params = list())
```

*Arguments:*

`x` Vector of points.

`log` Flag. If TRUE, return the gradient of the log-density instead.

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `length(x)`. In that case, the *i*-th density point will use the *i*-th parameters.

*Details:* Gradients of the density of a Distribution

*Returns:* A list structure containing the (log-)density gradients of all free parameters of the Distribution evaluated at `x`.

*Examples:*

```
dist_exponential()$diff_density(
  c(1.0, 2.0),
  with_params = list(rate = 2.0)
)
```

**Method** `diff_probability()`:

*Usage:*

```
Distribution$diff_probability(
  q,
  lower.tail = TRUE,
  log.p = FALSE,
  with_params = list()
)
```

*Arguments:*

`q` Vector of points to evaluate the probability function at.

`lower.tail` If TRUE, return  $P(X \leq q)$ . Otherwise return  $P(X > q)$ .

`log.p` If TRUE, probabilities are returned as  $\log(p)$ .

`with_params` Distribution parameters to use. Each parameter value can also be a numeric vector of length `length(q)`. In that case, the *i*-th probability point will use the *i*-th parameters.

*Details:* Gradients of the cumulative probability of a Distribution

*Returns:* A list structure containing the cumulative (log-)probability gradients of all free parameters of the Distribution evaluated at `q`.

*Examples:*

```
dist_exponential()$diff_probability(
  c(1.0, 2.0),
  with_params = list(rate = 2.0)
)
```

**Method is\_in\_support():***Usage:*

```
Distribution$is_in_support(x, with_params = list())
```

*Arguments:*

x Vector of points

with\_params Distribution parameters to use. Each parameter value can also be a numeric vector of length length(x). In that case, the i-th point will use the i-th parameters.

*Details:* Determine if a value is in the support of a Distribution

*Returns:* A logical vector with the same length as x indicating whether x is part of the support of the distribution given its parameters.

*Examples:*

```
dist_exponential(rate = 1.0)$is_in_support(c(-1.0, 0.0, 1.0))
```

**Method is\_discrete\_at():***Usage:*

```
Distribution$is_discrete_at(x, with_params = list())
```

*Arguments:*

x Vector of points

with\_params Distribution parameters to use. Each parameter value can also be a numeric vector of length length(x). In that case, the i-th point will use the i-th parameters.

*Details:* Determine if a value has positive probability

*Returns:* A logical vector with the same length as x indicating whether there is a positive probability mass at x given the Distribution parameters.

*Examples:*

```
dist_dirac(point = 0.0)$is_discrete_at(c(0.0, 1.0))
```

**Method tf\_is\_discrete\_at():***Usage:*

```
Distribution$tf_is_discrete_at()
```

*Details:* Compile a TensorFlow function for discrete support checking

*Returns:* A tf\_function taking arguments x and args returning whether the Distribution has a point mass at x given parameters args.

**Method has\_capability():***Usage:*

```
Distribution$has_capability(caps)
```

*Arguments:*

caps Character vector of capabilities

*Details:* Check if a capability is present

*Returns:* A logical vector the same length as caps.

*Examples:*

```
dist_exponential()$has_capability("density")
```

**Method** `get_type()`:

*Usage:*

```
Distribution$get_type()
```

*Details:* Get the type of a Distribution. Type can be one of discrete, continuous or mixed.

*Returns:* A string representing the type of the Distribution.

*Examples:*

```
dist_exponential()$get_type()
```

```
dist_dirac()$get_type()
```

```
dist_mixture(list(dist_dirac(), dist_exponential()))$get_type()
```

```
dist_mixture(list(dist_dirac(), dist_binomial()))$get_type()
```

**Method** `get_components()`:

*Usage:*

```
Distribution$get_components()
```

*Details:* Get the component Distributions of a transformed Distribution.

*Returns:* A possibly empty list of Distributions

*Examples:*

```
dist_trunc(dist_exponential())$get_components()
```

```
dist_dirac()$get_components()
```

```
dist_mixture(list(dist_exponential(), dist_gamma()))$get_components()
```

**Method** `is_discrete()`:

*Usage:*

```
Distribution$is_discrete()
```

*Details:* Check if a Distribution is discrete, i.e. it has a density with respect to the counting measure.

*Returns:* TRUE if the Distribution is discrete, FALSE otherwise. Note that mixed distributions are not discrete but can have point masses.

*Examples:*

```
dist_exponential()$is_discrete()
```

```
dist_dirac()$is_discrete()
```

**Method** `is_continuous()`:

*Usage:*

```
Distribution$is_continuous()
```

*Details:* Check if a Distribution is continuous, i.e. it has a density with respect to the Lebesgue measure.

*Returns:* TRUE if the Distribution is continuous, FALSE otherwise. Note that mixed distributions are not continuous.

*Examples:*

```
dist_exponential()$is_continuous()
dist_dirac()$is_continuous()
```

**Method** `require_capability()`:

*Usage:*

```
Distribution$require_capability(
  caps,
  fun_name = paste0(sys.call(-1)[[1]], "()")
)
```

*Arguments:*

`caps` Character vector of Capabilities to require

`fun_name` Friendly text to use for generating the error message in case of failure.

*Details:* Ensure that a Distribution has all required capabilities. Will throw an error if any capability is missing.

*Returns:* Invisibly TRUE.

*Examples:*

```
dist_exponential()$require_capability("diff_density")
```

**Method** `get_dof()`:

*Usage:*

```
Distribution$get_dof()
```

*Details:* Get the number of degrees of freedom of a Distribution family. Only parameters without a fixed default are considered free.

*Returns:* An integer representing the degrees of freedom suitable e.g. for AIC calculations.

*Examples:*

```
dist_exponential()$get_dof()
dist_exponential(rate = 1.0)$get_dof()
```

**Method** `get_placeholders()`:

*Usage:*

```
Distribution$get_placeholders()
```

*Details:* Get Placeholders of a Distribution family. Returns a list of free parameters of the family. Their values will be NULL.

If the Distribution has Distributions as parameters, placeholders will be computed recursively.

*Returns:* A named list containing any combination of (named or unnamed) lists and NULLs.

*Examples:*



```
dist_exponential().$get_placeholders()
dist_mixture(list(dist_dirac(), dist_exponential()))$get_placeholders()
```

**Method** `get_params()`:*Usage:*

```
Distribution$get_params(with_params = list())
```

*Arguments:*

`with_params` Optional parameter overrides with the same structure as `dist$get_params()`.  
Given Parameter values are expected to be length 1.

*Details:* Get a full list of parameters, possibly including placeholders.

*Returns:* A list representing the (recursive) parameter structure of the Distribution with values for specified parameters and NULL for free parameters that are missing both in the Distributions parameters and in `with_params`.

*Examples:*

```
dist_mixture(list(dist_dirac(), dist_exponential()))$get_params(
  with_params = list(probs = list(0.5, 0.5))
)
```

**Method** `tf_make_constants()`:*Usage:*

```
Distribution$tf_make_constants(with_params = list())
```

*Arguments:*

`with_params` Optional parameter overrides with the same structure as `dist$tf_make_constants()`.  
Given Parameter values are expected to be length 1.

*Details:* Get a list of constant TensorFlow parameters

*Returns:* A list representing the (recursive) constant parameters of the Distribution with values specified by parameters. Each constant is a TensorFlow Tensor of dtype floatx.

**Method** `tf_compile_params()`:*Usage:*

```
Distribution$tf_compile_params(input, name_prefix = "")
```

*Arguments:*

`input` A keras layer to bind all outputs to  
`name_prefix` Prefix to use for layer names

*Details:* Compile distribution parameters into tensorflow outputs

*Returns:* A list with two elements

- outputs a flat list of keras output layers, one for each parameter.
- `output_inflater` a function taking keras output layers and transforming them into a list structure suitable for passing to the loss function returned by `tf_compile_model()`

**Method** `get_param_bounds()`:*Usage:*

```
Distribution$get_param_bounds()
```

*Details:* Get Interval bounds on all Distribution parameters

*Returns:* A list representing the free (recursive) parameter structure of the Distribution with Interval objects as values representing the bounds of the respective free parameters.

*Examples:*

```
dist_mixture(
  list(dist_dirac(), dist_exponential()),
  probs = list(0.5, 0.5)
)$get_param_bounds()
```

```
dist_mixture(
  list(dist_dirac(), dist_exponential())
)$get_param_bounds()
```

```
dist_genpareto()$get_param_bounds()
dist_genpareto1()$get_param_bounds()
```

**Method** `get_param_constraints()`:

*Usage:*

```
Distribution$get_param_constraints()
```

*Details:* Get additional (non-linear) equality constraints on Distribution parameters

*Returns:* NULL if the box constraints specified by `dist$get_param_bounds()` are sufficient, or a function taking full Distribution parameters and returning either a numeric vector (which must be 0 for valid parameter combinations) or a list with elements

- `constraints`: The numeric vector of constraints
- `jacobian`: The Jacobi matrix of the constraints with respect to the parameters

*Examples:*

```
dist_mixture(
  list(dist_dirac(), dist_exponential())
)$get_param_constraints()
```

**Method** `export_functions()`:

*Usage:*

```
Distribution$export_functions(
  name,
  envir = parent.frame(),
  with_params = list()
)
```

*Arguments:*

`name` common suffix of the exported functions

`envir` Environment to export the functions to

`with_params` Optional list of parameters to use as default values for the exported functions

*Details:* Export sampling, density, probability and quantile functions to plain R functions. Creates new functions in envir named {r,d,p,q}<name> which implement `dist$sample`, `dist$density`, `dist$probability` and `dist$quantile` as plain functions with default arguments specified by `with_params` or the fixed parameters.

The resulting functions will have signatures taking all parameters as separate arguments.

*Returns:* Invisibly NULL.

*Examples:*

```
tmp_env <- new.env(parent = globalenv())
dist_exponential()$export_functions(
  name = "exp",
  envir = tmp_env,
  with_params = list(rate = 2.0)
)
evalq(
  fitdistrplus::fitdist(rexp(100), "exp"),
  envir = tmp_env
)
```

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

*Usage:*

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

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other Distributions: [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

## Examples

```
# Example for param_bounds:

# Create an Exponential Distribution with rate constrained to (0, 2)
# instead of (0, Inf)
my_exp <- dist_exponential()
my_exp$param_bounds$rate <- interval(c(0, 2))
my_exp$get_param_bounds()

fit_dist(my_exp, rexp(100, rate = 3), start = list(rate = 1))$params$rate

## -----
## Method `Distribution$sample`
## -----
```

```

dist_exponential(rate = 2.0)$sample(10)

## -----
## Method `Distribution$density`
## -----

dist_exponential()$density(c(1.0, 2.0), with_params = list(rate = 2.0))

## -----
## Method `Distribution$probability`
## -----

dist_exponential()$probability(
  c(1.0, 2.0),
  with_params = list(rate = 2.0)
)

## -----
## Method `Distribution$quantile`
## -----

dist_exponential()$quantile(c(0.1, 0.5), with_params = list(rate = 2.0))

## -----
## Method `Distribution$hazard`
## -----

dist_exponential(rate = 2.0)$hazard(c(1.0, 2.0))

## -----
## Method `Distribution$diff_density`
## -----

dist_exponential()$diff_density(
  c(1.0, 2.0),
  with_params = list(rate = 2.0)
)

## -----
## Method `Distribution$diff_probability`
## -----

dist_exponential()$diff_probability(
  c(1.0, 2.0),
  with_params = list(rate = 2.0)
)

## -----
## Method `Distribution$is_in_support`
## -----

dist_exponential(rate = 1.0)$is_in_support(c(-1.0, 0.0, 1.0))

```

```

## -----
## Method `Distribution$is_discrete_at`
## -----

dist_dirac(point = 0.0)$is_discrete_at(c(0.0, 1.0))

## -----
## Method `Distribution$has_capability`
## -----

dist_exponential()$has_capability("density")

## -----
## Method `Distribution$get_type`
## -----

dist_exponential()$get_type()
dist_dirac()$get_type()

dist_mixture(list(dist_dirac(), dist_exponential()))$get_type()
dist_mixture(list(dist_dirac(), dist_binomial()))$get_type()

## -----
## Method `Distribution$get_components`
## -----

dist_trunc(dist_exponential())$get_components()
dist_dirac()$get_components()
dist_mixture(list(dist_exponential(), dist_gamma()))$get_components()

## -----
## Method `Distribution$is_discrete`
## -----

dist_exponential()$is_discrete()
dist_dirac()$is_discrete()

## -----
## Method `Distribution$is_continuous`
## -----

dist_exponential()$is_continuous()
dist_dirac()$is_continuous()

## -----
## Method `Distribution$require_capability`
## -----

dist_exponential()$require_capability("diff_density")

## -----
## Method `Distribution$get_dof`
## -----

```

```

dist_exponential()$get_dof()
dist_exponential(rate = 1.0)$get_dof()

## -----
## Method `Distribution$get_placeholders`
## -----

dist_exponential()$get_placeholders()
dist_mixture(list(dist_dirac(), dist_exponential()))$get_placeholders()

## -----
## Method `Distribution$get_params`
## -----

dist_mixture(list(dist_dirac(), dist_exponential()))$get_params(
  with_params = list(probs = list(0.5, 0.5))
)

## -----
## Method `Distribution$get_param_bounds`
## -----

dist_mixture(
  list(dist_dirac(), dist_exponential()),
  probs = list(0.5, 0.5)
)$get_param_bounds()

dist_mixture(
  list(dist_dirac(), dist_exponential())
)$get_param_bounds()

dist_genpareto()$get_param_bounds()
dist_genpareto1()$get_param_bounds()

## -----
## Method `Distribution$get_param_constraints`
## -----

dist_mixture(
  list(dist_dirac(), dist_exponential())
)$get_param_constraints()

## -----
## Method `Distribution$export_functions`
## -----

tmp_env <- new.env(parent = globalenv())
dist_exponential()$export_functions(
  name = "exp",
  envir = tmp_env,
  with_params = list(rate = 2.0)
)

```

```
evalq(
  fitdistrplus::fitdist(rexp(100), "exp"),
  envir = tmp_env
)
```

---

dist_bdegp	<i>Construct a BDEGP-Family</i>
------------	---------------------------------

---

### Description

Constructs a BDEGP-Family distribution with fixed number of components and blending interval.

### Usage

```
dist_bdegp(n, m, u, epsilon)
```

### Arguments

n	Number of dirac components, starting with a point mass at 0.
m	Number of erlang components, translated by $n - 0.5$ .
u	Blending cut-off, must be a positive real.
epsilon	Blending radius, must be a positive real less than u. The blending interval will be $u - \text{epsilon} < x < u + \text{epsilon}$ .

### Value

- A MixtureDistribution of
  - n DiracDistributions at  $0 \dots n - 1$  and
  - a BlendedDistribution object with child Distributions
    - \* a TranslatedDistribution with offset  $n - 0.5$  of an ErlangMixtureDistribution with m shapes
    - \* and a GeneralizedParetoDistribution with shape parameter restricted to  $[0, 1]$  and location parameter fixed at u With break u and bandwidth epsilon.

### See Also

Other Distributions: [Distribution](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```

dist <- dist_bdegp(n = 1, m = 2, u = 10, epsilon = 3)
params <- list(
  dists = list(
    list(),
    list(
      dists = list(
        list(
          dist = list(
            shapes = list(1L, 2L),
            scale = 1.0,
            probs = list(0.7, 0.3)
          )
        ),
        list(
          sigmau = 1.0,
          xi = 0.1
        )
      ),
      probs = list(0.1, 0.9)
    )
  ),
  probs = list(0.95, 0.05)
)
x <- dist$sample(100, with_params = params)

plot_distributions(
  theoretical = dist,
  empirical = dist_empirical(x),
  .x = seq(0, 20, length.out = 101),
  with_params = list(theoretical = params)
)

```

---

dist\_beta

*Beta Distribution*


---

**Description**

See [stats::Beta](#)

**Usage**

```
dist_beta(shape1 = NULL, shape2 = NULL, ncp = NULL)
```

**Arguments**

shape1	First scalar shape parameter, or NULL as a placeholder.
shape2	Second scalar shape parameter, or NULL as a placeholder.
ncp	Scalar non-centrality parameter, or NULL as a placeholder.



**Details**

All parameters can be overridden with `with_params = list(shape = ..., scale = ...)`.

**Value**

A BetaDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_beta <- dist_beta(shape1 = 2, shape2 = 2, ncp = 0)
x <- d_beta$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_beta,
  estimated = d_beta,
  with_params = list(
    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "beta")$estimate
    )
  ),
  .x = seq(0, 2, length.out = 100)
)
```

---

dist_binomial	<i>Binomial Distribution</i>
---------------	------------------------------

---

**Description**

See [stats::Binomial](#)

**Usage**

```
dist_binomial(size = NULL, prob = NULL)
```

**Arguments**

size	Number of trials parameter (integer), or NULL as a placeholder.
prob	Success probability parameter, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(size = ..., prob = ...)`.

**Value**

A BinomialDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_bleded\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_binom <- dist_binomial(size = 10, prob = 0.5)
x <- d_binom$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_binom,
  estimated = d_binom,
  with_params = list(
    estimated = list(
      size = max(x),
      prob = mean(x) / max(x)
    )
  ),
  .x = 0:max(x)
)
```

---

dist\_bleded

*Blended distribution*

---

**Description**

Blended distribution

**Usage**

```
dist_bleded(dists, probs = NULL, breaks = NULL, bandwidths = NULL)
```

**Arguments**

dists	A list of $k \geq 2$ component Distributions.
probs	$k$ Mixture weight parameters
breaks	$k - 1$ Centers of the blending zones. <code>dists[i]</code> will blend into <code>dists[i + 1]</code> around <code>breaks[i]</code> .
bandwidths	$k - 1$ Radii of the blending zones. The $i$ -th blending zone will begin at <code>breaks[i] - bandwidths[i]</code> and end at <code>breaks[i] + bandwidths[i]</code> . A bandwidth of 0 corresponds to a hard cut-off, i.e. a jump discontinuity in the density of the blended Distribution.

**Value**

A `BlendedDistribution` object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
bd <- dist_blended(
  list(
    dist_normal(mean = 0.0, sd = 1.0),
    dist_genpareto(u = 3.0, sigmau = 1.0, xi = 3.0)
  ),
  breaks = list(3.0),
  bandwidths = list(0.5),
  probs = list(0.9, 0.1)
)

plot_distributions(
  bd,
  .x = seq(-3, 10, length.out = 100),
  plots = c("d", "p")
)
```

---

 dist\_dirac

*Dirac (degenerate point) Distribution*


---

**Description**

A degenerate distribution with all mass at a single point.

**Usage**

```
dist_dirac(point = NULL)
```

**Arguments**

point            The point with probability mass 1.

**Details**

The parameter can be overridden with `with_params = list(point = ...)`.

**Value**

A DiracDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_dirac <- dist_dirac(1.5)
d_dirac$sample(2L)
d_dirac$sample(2L, list(point = 42.0))
```

---

dist_discrete	<i>Discrete Distribution</i>
---------------	------------------------------

---

**Description**

A full-flexibility discrete distribution with values from 1 to size.

**Usage**

```
dist_discrete(size = NULL, probs = NULL)
```

**Arguments**

size            Number of classes parameter (integer). Required if probs is NULL.  
 probs          Vector of probabilities parameter, or NULL as a placeholder.

**Details**

Parameters can be overridden with `with_params = list(probs = ...)`.

**Value**

A DiscreteDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_discrete <- dist_discrete(probs = list(0.5, 0.25, 0.15, 0.1))
x <- d_discrete$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_discrete,
  estimated = d_discrete,
  with_params = list(
    estimated = list(
      size = max(x),
      probs = as.list(unname(table(x)) / 100)
    )
  ),
  .x = 0:max(x)
)
```

---

dist_empirical	<i>Empirical distribution</i>
----------------	-------------------------------

---

**Description**

Creates an empirical distribution object from a sample. Assumes iid. samples. with\_params should **not** be used with this distribution because estimation of the relevant indicators happens during construction.

**Usage**

```
dist_empirical(sample, positive = FALSE, bw = "nrd0")
```

**Arguments**

sample                      Sample to build the empirical distribution from

positive	Is the underlying distribution known to be positive? This will effect the density estimation procedure. <code>positive = FALSE</code> uses a kernel density estimate produced by <code>density()</code> , <code>positive = TRUE</code> uses a log-kernel density estimate produced by <code>logKDE::logdensity_fft()</code> . The latter can improve density estimation near zero.
bw	Bandwidth parameter for density estimation. Passed to the density estimation function selected by <code>positive</code> .

### Details

- `sample()` samples iid. from `sample`. This approach is similar to bootstrapping.
- `density()` evaluates a kernel density estimate, approximating with zero outside of the known support. This estimate is either obtained using `stats::density` or `logKDE::logdensity_fft`, depending on `positive`.
- `probability()` evaluates the empirical cumulative density function obtained by `stats::ecdf`.
- `quantile()` evaluates the empirical quantiles using `stats::quantile`
- `hazard()` estimates the hazard rate using the density estimate and the empirical cumulative density function:  $h(t) = df(t) / (1 - cdf(t))$ .

### Value

An `EmpiricalDistribution` object.

### See Also

Other Distributions: `Distribution`, `dist_bdegn`(), `dist_beta`(), `dist_binomial`(), `dist_blended`(), `dist_dirac`(), `dist_discrete`(), `dist_erlangmix`(), `dist_exponential`(), `dist_gamma`(), `dist_genpareto`(), `dist_lognormal`(), `dist_mixture`(), `dist_negbinomial`(), `dist_normal`(), `dist_pareto`(), `dist_poisson`(), `dist_translate`(), `dist_trunc`(), `dist_uniform`(), `dist_weibull`()

### Examples

```
x <- rexp(20, rate = 1)
dx <- dist_empirical(sample = x, positive = TRUE)

y <- rnorm(20)
dy <- dist_empirical(sample = y)

plot_distributions(
  exponential = dx,
  normal = dy,
  .x = seq(-3, 3, length.out = 100)
)
```

---

dist_erlangmix	<i>Erlang Mixture distribution</i>
----------------	------------------------------------

---

## Description

Erlang Mixture distribution

## Usage

```
dist_erlangmix(shapes, scale = NULL, probs = NULL)
```

## Arguments

shapes	Shape parameters, a trunc_erlangmix fit, or NULL as a placeholder.
scale	Common scale parameter, or NULL as a placeholder.
probs	Mixing probabilities, or NULL as a placeholder.

## Value

An ErlangMixtureDistribution object.

## See Also

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

## Examples

```
params <- list(scale = 1.0, probs = list(0.5, 0.3, 0.2), shapes = list(1L, 2L, 3L))
dist <- dist_erlangmix(vector("list", 3L))
x <- dist$sample(20, with_params = params)
d_emp <- dist_empirical(x, positive = TRUE)

plot_distributions(
  empirical = d_emp,
  theoretical = dist,
  with_params = list(
    theoretical = params
  ),
  .x = seq(1e-4, 5, length.out = 100)
)
```

---

dist_exponential	<i>Exponential distribution</i>
------------------	---------------------------------

---

**Description**

See [stats::Exponential](#).

**Usage**

```
dist_exponential(rate = NULL)
```

**Arguments**

rate                    Scalar rate parameter, or NULL as a placeholder.

**Details**

The parameter can be overridden with `with_params = list(rate = ...)`.

**Value**

An `ExponentialDistribution` object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
rate <- 1
d_exp <- dist_exponential()
x <- d_exp$sample(20, with_params = list(rate = rate))
d_emp <- dist_empirical(x, positive = TRUE)

plot_distributions(
  empirical = d_emp,
  theoretical = d_exp,
  estimated = d_exp,
  with_params = list(
    theoretical = list(rate = rate),
    estimated = list(rate = 1 / mean(x))
  ),
  .x = seq(1e-4, 5, length.out = 100)
)
```



---

`dist_gamma`*Gamma distribution*

---

**Description**

See [stats::GammaDist](#).

**Usage**

```
dist_gamma(shape = NULL, rate = NULL)
```

**Arguments**

<code>shape</code>	Scalar shape parameter, or NULL as a placeholder.
<code>rate</code>	Scalar rate parameter, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(shape = ..., rate = ...)`.

**Value**

A GammaDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegn\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
alpha <- 2
beta <- 2

d_gamma <- dist_gamma(shape = alpha, rate = beta)
x <- d_gamma$sample(100)
d_emp <- dist_empirical(x, positive = TRUE)

plot_distributions(
  empirical = d_emp,
  theoretical = d_gamma,
  estimated = d_gamma,
  with_params = list(
    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "gamma")$estimate
    )
  ),
)
```

```
.x = seq(1e-3, max(x), length.out = 100)
)
```

---

dist_genpareto	<i>Generalized Pareto Distribution</i>
----------------	--

---

### Description

See [evmix::gpd](#)

### Usage

```
dist_genpareto(u = NULL, sigmau = NULL, xi = NULL)
```

```
dist_genpareto1(u = NULL, sigmau = NULL, xi = NULL)
```

### Arguments

u	Scalar location parameter, or NULL as a placeholder.
sigmau	Scalar scale parameter, or NULL as a placeholder.
xi	Scalar shape parameter, or NULL as a placeholder.

### Details

All parameters can be overridden with `with_params = list(u = ..., sigmau = ..., xi = ...)`.

`dist_genpareto1` is equivalent to `dist_genpareto` but enforces bound constraints on `xi` to  $[\emptyset, 1]$ . This ensures unboundedness and finite expected value unless `xi == 1.0`.

### Value

A `GeneralizedParetoDistribution` object.

### See Also

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```

d_genpareto <- dist_genpareto(u = 0, sigmau = 1, xi = 1)
x <- d_genpareto$sample(100)
d_emp <- dist_empirical(x)

d_genpareto$export_functions("gpd") # so fitdistrplus finds it

plot_distributions(
  empirical = d_emp,
  theoretical = d_genpareto,
  estimated = d_genpareto,
  with_params = list(
    estimated = fit(dist_genpareto(), x)$params
  ),
  .x = seq(0, 5, length.out = 100)
)

```

---

dist_lognormal	<i>Log Normal distribution</i>
----------------	--------------------------------

---

**Description**

See [stats::Lognormal](#).

**Usage**

```
dist_lognormal(meanlog = NULL, sdlog = NULL)
```

**Arguments**

meanlog            Scalar mean parameter on the log scale, or NULL as a placeholder.  
sdlog                Scalar standard deviation parameter on the log scale, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(meanlog = ..., sdlog = ...)`.

**Value**

A LognormalDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```

mu <- 0
sigma <- 1

d_lnorm <- dist_lognormal(meanlog = mu, sdlog = sigma)
x <- d_lnorm$sample(20)
d_emp <- dist_empirical(x, positive = TRUE)

plot_distributions(
  empirical = d_emp,
  theoretical = d_lnorm,
  estimated = d_lnorm,
  with_params = list(
    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "lnorm")$estimate
    )
  ),
  .x = seq(1e-3, 5, length.out = 100)
)

```

---

dist\_mixture

*Mixture distribution*


---

**Description**

Parameters of mixing components can be overridden with `with_params = list(dists = list(..., ..., ...))`. #' Mixing probabilities can be overridden with `with_params = list(probs = list(..., ..., ...))`. The **number of components** cannot be overridden.

**Usage**

```
dist_mixture(dists = list(), probs = NULL)
```

**Arguments**

dists	A list of mixing distributions. May contain placeholders and duplicates.
probs	A list of mixing probabilities with the same length as <code>dists</code> . They are normalized to sum to one and <code>NULL</code> can be used as a placeholder within <code>probs</code> . To reduce the number of required parameters, <code>probs</code> should at least be partly specified ( <code>probs = list(NULL, NULL, ..., 1)</code> with <code>k - 1</code> <code>NULL</code> s where <code>k</code> is the number of mixing components).

**Details**

Does **not** support the `quantile()` capability!

**Value**

A MixtureDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
# A complicated way to define a uniform distribution on \[0, 2\]
dist_mixture(
  dists = list(
    dist_uniform(min = 0, max = 1),
    dist_uniform(min = 1, max = 2)
  ),
  probs = list(0.5, 0.5)
)
```

---

dist_negbinomial	<i>Negative binomial Distribution</i>
------------------	---------------------------------------

---

**Description**

See [stats::NegBinomial](#)

**Usage**

```
dist_negbinomial(size = NULL, mu = NULL)
```

**Arguments**

size	Number of successful trials parameter, or NULL as a placeholder. Non-integer values > 0 are allowed.
mu	Mean parameter, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(size = ..., prob = ...)`.

**Value**

A NegativeBinomialDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegg\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_nbinom <- dist_negbinomial(size = 3.5, mu = 8.75)
x <- d_nbinom$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_nbinom,
  estimated = d_nbinom,
  with_params = list(
    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "nbinom")$estimate
    )
  ),
  .x = 0:max(x)
)
```

---

 dist\_normal

*Normal distribution*


---

**Description**

See [stats::Normal](#).

**Usage**

```
dist_normal(mean = NULL, sd = NULL)
```

**Arguments**

mean	Scalar mean parameter, or NULL as a placeholder.
sd	Scalar standard deviation parameter, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(mean = ..., sd = ...)`.

**Value**

A `NormalDistribution` object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
mu <- 0
sigma <- 1

d_norm <- dist_normal(mean = mu, sd = sigma)
x <- d_norm$sample(20)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_norm,
  estimated = d_norm,
  with_params = list(
    estimated = list(mean = mean(x), sd = sd(x))
  ),
  .x = seq(-3, 3, length.out = 100)
)
```

---

dist\_pareto

*Pareto Distribution*

---

**Description**

See [Pareto](#)

**Usage**

```
dist_pareto(shape = NULL, scale = NULL)
```

**Arguments**

shape	Scalar shape parameter, or NULL as a placeholder.
scale	Scalar scale parameter, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(shape = ..., scale = ...)`.

**Value**

A `ParetoDistribution` object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_pareto <- dist_pareto(shape = 3, scale = 1)
x <- d_pareto$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_pareto,
  estimated = d_pareto,
  with_params = list(
    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "pareto")$estimate
    )
  ),
  .x = seq(0, 2, length.out = 100)
)
```

---

 dist\_poisson

*Poisson Distribution*


---

**Description**

See [stats::Poisson](#)

**Usage**

```
dist_poisson(lambda = NULL)
```

**Arguments**

lambda            Scalar rate parameter, or NULL as a placeholder.

**Details**

The parameter can be overridden with `with_params = list(lambda = ...)`.

**Value**

A `PoissonDistribution` object.



**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_pois <- dist_poisson(lambda = 5.0)
x <- d_pois$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_pois,
  estimated = d_pois,
  with_params = list(
    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "pois")$estimate
    )
  ),
  .x = 0:max(x)
)
```

---

dist_translate	<i>Translated distribution</i>
----------------	--------------------------------

---

**Description**

Translated distribution

**Usage**

```
dist_translate(dist = NULL, offset = NULL, multiplier = 1)
```

**Arguments**

dist	An underlying distribution, or NULL as a placeholder.
offset	Offset to be added to each observation, or NULL as a placeholder.
multiplier	Factor to multiply each observation by, or NULL as a placeholder.

**Value**

A TranslatedDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_norm <- dist_normal(mean = 0, sd = 1)
d_tnorm <- dist_translate(dist = d_norm, offset = 1)
plot_distributions(d_norm, d_tnorm, .x = seq(-2, 3, length.out = 100))
```

---

dist_trunc	<i>Truncated distribution</i>
------------	-------------------------------

---

**Description**

Truncated distribution

**Usage**

```
dist_trunc(dist = NULL, min = NULL, max = NULL, offset = 0, max_retry = 100)
```

**Arguments**

<code>dist</code>	An underlying distribution, or NULL as a placeholder.
<code>min</code>	Minimum value to truncate at (exclusive), or NULL as a placeholder.
<code>max</code>	Maximum value to truncate at (inclusive), or NULL as a placeholder.
<code>offset</code>	Offset to be added to each observation after truncation, or NULL as a placeholder. Truncation of <code>dist</code> will occur to <code>(min, max]</code> . The offset is then added deterministically.
<code>max_retry</code>	Maximum number of resample attempts when trying to sample with rejection.

**Value**

A TruncatedDistribution object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_uniform\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_norm <- dist_normal(mean = 0, sd = 1)
d_tnorm <- dist_trunc(dist = d_norm, min = -2, max = 2, offset = 1)
plot_distributions(d_norm, d_tnorm, .x = seq(-2, 3, length.out = 100))
```

---

dist_uniform	<i>Uniform distribution</i>
--------------	-----------------------------

---

**Description**

See [stats::Uniform](#)

**Usage**

```
dist_uniform(min = NULL, max = NULL)
```

**Arguments**

min	Lower limit, or NULL as a placeholder.
max	Upper limit, or NULL as a placeholder.

**Details**

Both parameters can be overridden with `with_params = list(min = ..., max = ...)`.

**Value**

A `UniformDistribution` object.

**See Also**

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_weibull\(\)](#)

**Examples**

```
d_unif <- dist_uniform(min = 0, max = 1)
x <- d_unif$sample(100)
d_emp <- dist_empirical(x)
```

```
plot_distributions(
  empirical = d_emp,
  theoretical = d_unif,
  estimated = d_unif,
  with_params = list(
```

```

    estimated = inflate_params(
      fitdistrplus::fitdist(x, distr = "unif")$estimate
    )
  ),
  .x = seq(0, 1, length.out = 100)
)

```

---

 dist\_weibull

*Weibull Distribution*


---

### Description

See [stats::Weibull](#)

### Usage

```
dist_weibull(shape = NULL, scale = NULL)
```

### Arguments

shape            Scalar shape parameter, or NULL as a placeholder.  
 scale            Scalar scale parameter, or NULL as a placeholder.

### Details

Both parameters can be overridden with `with_params = list(shape = ..., scale = ...)`.

### Value

A WeibullDistribution object.

### See Also

Other Distributions: [Distribution](#), [dist\\_bdegp\(\)](#), [dist\\_beta\(\)](#), [dist\\_binomial\(\)](#), [dist\\_blended\(\)](#), [dist\\_dirac\(\)](#), [dist\\_discrete\(\)](#), [dist\\_empirical\(\)](#), [dist\\_erlangmix\(\)](#), [dist\\_exponential\(\)](#), [dist\\_gamma\(\)](#), [dist\\_genpareto\(\)](#), [dist\\_lognormal\(\)](#), [dist\\_mixture\(\)](#), [dist\\_negbinomial\(\)](#), [dist\\_normal\(\)](#), [dist\\_pareto\(\)](#), [dist\\_poisson\(\)](#), [dist\\_translate\(\)](#), [dist\\_trunc\(\)](#), [dist\\_uniform\(\)](#)

### Examples

```

d_weibull <- dist_weibull(shape = 3, scale = 1)
x <- d_weibull$sample(100)
d_emp <- dist_empirical(x)

plot_distributions(
  empirical = d_emp,
  theoretical = d_weibull,
  estimated = d_weibull,

```

```

with_params = list(
  estimated = inflate_params(
    fitdistrplus::fitdist(x, distr = "weibull")$estimate
  )
),
.x = seq(0, 2, length.out = 100)
)

```

---

```
fit.reservr_keras_model
```

*Fit a neural network based distribution model to data*

---

### Description

This function delegates most work to `keras::fit.keras.engine.training.Model()` and performs additional consistency checks to make sure `tf_compile_model()` was called with the appropriate options to support fitting the observations `y` as well as automatically converting `y` to a `n x 6` matrix needed by the compiled loss function.

### Usage

```

## S3 method for class 'reservr_keras_model'
fit(
  object,
  x,
  y,
  batch_size = NULL,
  epochs = 10,
  verbose = getOption("keras.fit_verbose", default = 1),
  callbacks = NULL,
  view_metrics = getOption("keras.view_metrics", default = "auto"),
  validation_split = 0,
  validation_data = NULL,
  shuffle = TRUE,
  class_weight = NULL,
  sample_weight = NULL,
  initial_epoch = 0,
  steps_per_epoch = NULL,
  validation_steps = NULL,
  ...
)

```

### Arguments

<code>object</code>	A compiled <code>reservr_keras_model</code> as obtained by <code>tf_compile_model()</code> .
<code>x</code>	A list of input tensors (predictors)

<code>y</code>	A <code>trunc_obs</code> tibble of observed outcomes, or something convertible via <code>as_trunc_obs()</code> .
<code>batch_size</code>	Integer or NULL. Number of samples per gradient update. If unspecified, <code>batch_size</code> will default to 32.
<code>epochs</code>	Number of epochs to train the model. Note that in conjunction with <code>initial_epoch</code> , <code>epochs</code> is to be understood as "final epoch". The model is not trained for a number of iterations given by <code>epochs</code> , but merely until the epoch of index <code>epochs</code> is reached.
<code>verbose</code>	Verbosity mode (0 = silent, 1 = progress bar, 2 = one line per epoch).
<code>callbacks</code>	List of callbacks to be called during training.
<code>view_metrics</code>	View realtime plot of training metrics (by epoch). The default ("auto") will display the plot when running within RStudio, metrics were specified during <code>model_compile()</code> , <code>epochs &gt; 1</code> and <code>verbose &gt; 0</code> . Use the global <code>keras.view_metrics</code> option to establish a different default.
<code>validation_split</code>	Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the <code>x</code> and <code>y</code> data provided, before shuffling.
<code>validation_data</code>	Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. This could be a list ( <code>x_val</code> , <code>y_val</code> ) or a list ( <code>x_val</code> , <code>y_val</code> , <code>val_sample_weights</code> ). <code>validation_data</code> will override <code>validation_split</code> .
<code>shuffle</code>	<code>shuffle</code> : Logical (whether to shuffle the training data before each epoch) or string (for "batch"). "batch" is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when <code>steps_per_epoch</code> is not NULL.
<code>class_weight</code>	Optional named list mapping indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
<code>sample_weight</code>	Optional array of the same length as <code>x</code> , containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape ( <code>samples</code> , <code>sequence_length</code> ), to apply a different weight to every timestep of every sample. In this case you should make sure to specify <code>sample_weight_mode="temporal"</code> in <code>compile()</code> .
<code>initial_epoch</code>	Integer, Epoch at which to start training (useful for resuming a previous training run).
<code>steps_per_epoch</code>	Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default NULL is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined.
<code>validation_steps</code>	Only relevant if <code>steps_per_epoch</code> is specified. Total number of steps (batches of samples) to validate before stopping.

... Unused. If old arguments are supplied, an error message will be raised informing how to fix the issue.

### Details

Additionally, the default `batch_size` is `min(nrow(y), 10000)` instead of keras default of 32 because the latter is a very bad choice for fitting most distributions since the involved loss is much less stable than typical losses used in machine learning, leading to divergence for small batch sizes.

### Value

A history object that contains all information collected during training. The model object will be updated in-place as a side-effect.

### See Also

`predict.reservr_keras_model` `tf_compile_model` `keras::fit.keras.engine.training.Model`

### Examples

```
dist <- dist_exponential()
params <- list(rate = 1.0)
N <- 100L
rand_input <- runif(N)
x <- dist$sample(N, with_params = params)

if (keras::is_keras_available()) {
  tf_in <- keras::layer_input(1L)
  mod <- tf_compile_model(
    inputs = list(tf_in),
    intermediate_output = tf_in,
    dist = dist,
    optimizer = keras::optimizer_adam(),
    censoring = FALSE,
    truncation = FALSE
  )

  tf_fit <- fit(
    object = mod,
    x = k_matrix(rand_input),
    y = x,
    epochs = 10L,
    callbacks = list(
      callback_debug_dist_gradients(mod, k_matrix(rand_input), x, keep_grads = TRUE)
    )
  )
}
```

fit\_blen

*Fit a Blended mixture using an ECME-Algorithm***Description**

Fit a Blended mixture using an ECME-Algorithm

**Usage**

```
fit_blen(
  dist,
  obs,
  start,
  min_iter = 0L,
  max_iter = 100L,
  skip_first_e = FALSE,
  tolerance = 1e-05,
  trace = FALSE,
  ...
)
```

**Arguments**

dist	A BlendedDistribution. It is assumed, that breaks and bandwidths are not a placeholder and that weights are to be estimated.
obs	Set of observations as produced by <code>trunc_obs()</code> or convertible via <code>as_trunc_obs()</code> .
start	Initial values of all placeholder parameters. If missing, starting values are obtained from <code>fit_dist_start()</code> .
min_iter	Minimum number of EM-Iterations
max_iter	Maximum number of EM-Iterations (weight updates)
skip_first_e	Skip the first E-Step (update Probability weights)? This can help if the initial values cause a mixture component to vanish in the first E-Step before the starting values can be improved.
tolerance	Numerical tolerance.
trace	Include tracing information in output? If TRUE, additional tracing information will be added to the result list.
...	Passed to <code>fit_dist_start()</code> if start is missing.

**Value**

A list with elements

- `params` the fitted parameters in the same structure as `init`.
- `params_hist` (if `trace` is TRUE) the history of parameters (after each e- and m- step)
- `iter` the number of outer EM-iterations
- `logLik` the final log-likelihood



**See Also**

Other distribution fitting functions: [fit\\_dist\(\)](#), [fit\\_erlang\\_mixture\(\)](#), [fit\\_mixture\(\)](#)

**Examples**

```
dist <- dist_bleneded(
  list(
    dist_exponential(),
    dist_genpareto()
  )
)

params <- list(
  probs = list(0.9, 0.1),
  dists = list(
    list(rate = 2.0),
    list(u = 1.5, xi = 0.2, sigmau = 1.0)
  ),
  breaks = list(1.5),
  bandwidths = list(0.3)
)

x <- dist$sample(100L, with_params = params)

dist$default_params$breaks <- params$breaks
dist$default_params$bandwidths <- params$bandwidths
if (interactive()) {
  fit_bleneded(dist, x)
}
```

---

fit\_dist

*Fit a general distribution to observations*


---

**Description**

The default implementation performs maximum likelihood estimation on all placeholder parameters.

**Usage**

```
fit_dist(dist, obs, start, ...)

fit_dist_direct(dist, obs, start, ..., .start_with_default = FALSE)

## S3 method for class 'Distribution'
fit(object, obs, start, ...)
```

**Arguments**

dist	A Distribution object.
obs	Set of observations as produced by <code>trunc_obs()</code> or convertible via <code>as_trunc_obs()</code> .
start	Initial values of all placeholder parameters. If missing, starting values are obtained from <code>fit_dist_start()</code> .
...	Distribution-specific arguments for the fitting procedure
.start_with_default	Before directly optimising the likelihood, use an optimised algorithm for finding better starting values?
object	same as parameter dist

**Details**

For Erlang mixture distributions and for Mixture distributions, an EM-Algorithm is instead used to improve stability.

`fit()` and `fit_dist()` will chose an optimisation method optimized for the specific distribution given. `fit_dist_direct()` can be used to force direct maximisation of the likelihood.

**Value**

A list with at least the elements

- `params` the fitted parameters in the same structure as `init`.
- `logLik` the final log-likelihood

Additional information may be provided depending on `dist`.

**See Also**

Other distribution fitting functions: `fit_blended()`, `fit_erlang_mixture()`, `fit_mixture()`

Other distribution fitting functions: `fit_blended()`, `fit_erlang_mixture()`, `fit_mixture()`

**Examples**

```
x <- rexp(100)
lambda_hat <- 1 / mean(x)
lambda_hat2 <- fit_dist(dist_exponential(), x)$params$rate
identical(lambda_hat, lambda_hat2)
dist <- dist_mixture(list(dist_normal(), dist_translate(dist_exponential(), offset = 6)))
params <- list(
  dists = list(list(mean = 5, sd = 1), list(dist = list(rate = 1))), probs = list(0.95, 0.05)
)
set.seed(2000)
u <- runif(100, 10, 20)
x <- dist$sample(100, with_params = params)
obs <- trunc_obs(x = x[x <= u], tmin = -Inf, tmax = u[x <= u])

default_fit <- fit_dist(dist, obs)
```

```
direct_fit <- fit_dist_direct(dist, obs)
# NB: direct optimisation steps with pre-run take a few seconds

direct_fit_init <- fit_dist_direct(dist, obs, start = default_fit$params)
direct_fit_auto_init <- fit_dist_direct(dist, obs, .start_with_default = TRUE)

stopifnot(direct_fit_init$logLik == direct_fit_auto_init$logLik)

c(default_fit$logLik, direct_fit$logLik, direct_fit_init$logLik)
```

---

fit\_dist\_start.MixtureDistribution

*Find starting values for distribution parameters*

---

## Description

Find starting values for distribution parameters

## Usage

```
## S3 method for class 'MixtureDistribution'
fit_dist_start(dist, obs, dists_start = NULL, ...)

fit_dist_start(dist, obs, ...)
```

## Arguments

dist	A Distribution object.
obs	Observations to fit to.
dists_start	List of initial parameters for all component distributions. If left empty, initialisation will be automatically performed using <code>fit_dist_start()</code> with all observations in the support of each respective component.
...	Additional arguments for the initialisation procedure

## Value

A list of initial parameters suitable for passing to `fit_dist()`.

## Examples

```
fit_dist_start(dist_exponential(), rexp(100))
```

---

fit\_erlang\_mixture      *Fit an Erlang mixture using an ECME-Algorithm*

---

## Description

Fit an Erlang mixture using an ECME-Algorithm

## Usage

```
fit_erlang_mixture(
  dist,
  obs,
  start,
  min_iter = 0L,
  max_iter = 100L,
  skip_first_e = FALSE,
  tolerance = 1e-05,
  trace = FALSE,
  parallel = FALSE,
  ...
)
```

## Arguments

dist	An ErlangMixtureDistribution. It is assumed, that both probs and scale are to be estimated.
obs	Set of observations as produced by <code>trunc_obs()</code> or convertible via <code>as_trunc_obs()</code> .
start	Initial values of all placeholder parameters. If missing, starting values are obtained from <code>fit_dist_start()</code> .
min_iter	Minimum number of EM-Iterations
max_iter	Maximum number of EM-Iterations (weight updates)
skip_first_e	Skip the first E-Step (update Probability weights)? This can help if the initial values cause a mixture component to vanish in the first E-Step before the starting values can be improved.
tolerance	Numerical tolerance.
trace	Include tracing information in output? If TRUE, additional tracing information will be added to the result list.
parallel	Enable experimental parallel evaluation of expected log-likelihood?
...	Passed to <code>fit_dist_start()</code> if start is missing.

**Value**

A list with elements

- `params` the fitted parameters in the same structure as `init`.
- `params_hist` (if `trace` is `TRUE`) the history of parameters (after each e- and m- step). Otherwise an empty list.
- `iter` the number of outer EM-iterations
- `logLik` the final log-likelihood

**See Also**

Other distribution fitting functions: [fit\\_blended\(\)](#), [fit\\_dist\(\)](#), [fit\\_mixture\(\)](#)

**Examples**

```
dist <- dist_erlangmix(list(NULL, NULL, NULL))
params <- list(
  shapes = list(1L, 4L, 12L),
  scale = 2.0,
  probs = list(0.5, 0.3, 0.2)
)
x <- dist$sample(100L, with_params = params)
fit_erlang_mixture(dist, x, init = "kmeans")
```

---

fit\_mixture

*Fit a generic mixture using an ECME-Algorithm*

---

**Description**

Fit a generic mixture using an ECME-Algorithm

**Usage**

```
fit_mixture(
  dist,
  obs,
  start,
  min_iter = 0L,
  max_iter = 100L,
  skip_first_e = FALSE,
  tolerance = 1e-05,
  trace = FALSE,
  ...
)
```

**Arguments**

<code>dist</code>	A MixtureDistribution specifying the structure of the mixture. Free parameters are to be optimised. The dominating measure for likelihoods must be constant, so for example <code>dist_dirac()</code> may not have its point parameter free.
<code>obs</code>	Set of observations as produced by <code>trunc_obs()</code> or convertible via <code>as_trunc_obs()</code> .
<code>start</code>	Initial values of all placeholder parameters. If missing, starting values are obtained from <code>fit_dist_start()</code> .
<code>min_iter</code>	Minimum number of EM-Iterations
<code>max_iter</code>	Maximum number of EM-Iterations (weight updates)
<code>skip_first_e</code>	Skip the first E-Step (update Probability weights)? This can help if the initial values cause a mixture component to vanish in the first E-Step before the starting values can be improved.
<code>tolerance</code>	Numerical tolerance.
<code>trace</code>	Include tracing information in output? If TRUE, additional tracing information will be added to the result list.
<code>...</code>	Passed to <code>fit_dist_start()</code> if <code>start</code> is missing.

**Value**

A list with elements

- `params` the fitted parameters in the same structure as `init`.
- `params_hist` (if `trace` is TRUE) the history of parameters (after each e- and m- step)
- `iter` the number of outer EM-iterations
- `logLik` the final log-likelihood

**See Also**

Other distribution fitting functions: `fit_blended()`, `fit_dist()`, `fit_erlang_mixture()`

**Examples**

```
dist <- dist_mixture(
  list(
    dist_dirac(0.0),
    dist_exponential()
  )
)

params <- list(
  probs = list(0.1, 0.9),
  dists = list(
    list(),
    list(rate = 1.0)
  )
)
```

```
x <- dist$sample(100L, with_params = params)
fit_mixture(dist, x)
```

---

flatten\_params                      *Flatten / Inflate parameter lists / vectors*

---

### Description

Flatten / Inflate parameter lists / vectors

### Usage

```
flatten_params(params)
flatten_params_matrix(params)
flatten_bounds(bounds)
inflate_params(flat_params)
```

### Arguments

params	A named list of parameters to be flattened. Should be in a form to be passed as the with_params argument to most distribution functions.
bounds	List of parameter bounds as returned by dist\$get_param_bounds()
flat_params	A named numeric vector of parameters

### Value

flatten\_params returns a 'flattened' vector of parameters. It is intended as an adapter for multi-dimensional optimisation functions to distribution objects.

flatten\_params\_matrix returns a 'flattened' matrix of parameters. It is intended as an adapter for multi-dimensional optimisation functions to distribution objects. Each column corresponds to one input element.

flatten\_bounds returns a named list of vectors with names lower and upper. Containing the upper and lower bounds of each parameter.

inflate\_params returns an 'inflated' list of parameters. This can be passed as the with\_params argument to most distribution functions.

**Examples**

```

library(ggplot2)

mm <- dist_mixture(list(
  dist_exponential(NULL),
  dist_lognormal(0.5, NULL)
), list(NULL, 1))

ph <- mm$get_placeholders()
ph_flat <- flatten_params(ph)
ph_reinflated <- inflate_params(ph_flat)
ph_flat[] <- c(1, 1, 6)
ph_sample <- inflate_params(ph_flat)

x <- mm$sample(
  100,
  with_params = ph_sample
)

emp_cdf <- ecdf(x)

ggplot(data.frame(t = seq(from = min(x), to = max(x), length.out = 100))) %+%
  geom_point(aes(x = t, y = emp_cdf(t))) %+%
  geom_line(aes(x = t, y = mm$probability(t, with_params = ph_sample)),
    linetype = 2)

```

---

GenPareto

*The Generalized Pareto Distribution (GPD)*


---

**Description**

These functions provide information about the generalized Pareto distribution with threshold  $u$ . `dgpd` gives the density, `pgpd` gives the distribution function, `qgpd` gives the quantile function and `rgpd` generates random deviates.

**Usage**

```

rgpd(n = 1L, u = 0, sigmau = 1, xi = 0)

dgpd(x, u = 0, sigmau = 1, xi = 0, log = FALSE)

pgpd(q, u = 0, sigmau = 1, xi = 0, lower.tail = TRUE, log.p = FALSE)

qgpd(p, u = 0, sigmau = 1, xi = 0, lower.tail = TRUE, log.p = FALSE)

```

**Arguments**

`n` integer number of observations.



<code>u</code>	threshold parameter (minimum value).
<code>sigmau</code>	scale parameter (must be positive).
<code>xi</code>	shape parameter
<code>x, q</code>	vector of quantiles.
<code>log, log.p</code>	logical; if TRUE, probabilities/densities $p$ are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P(X \leq x)$ , otherwise $P(X > x)$ .
<code>p</code>	vector of probabilities.

### Details

If `u`, `sigmau` or `xi` are not specified, they assume the default values of 0, 1 and 0 respectively.

The generalized Pareto distribution has density

$$f(x) = 1/\sigma_u(1 + \xi z)^{-1/\xi - 1}$$

where  $z = (x - u)/\sigma_u$  and  $f(x) = \exp(-z)$  if  $\xi$  is 0. The support is  $x \geq u$  for  $\xi \geq 0$  and  $u \leq x \leq u - \sigma_u/\xi$  for  $\xi < 0$ .

The Expected value exists if  $\xi < 1$  and is equal to

$$E(X) = u + \sigma_u/(1 - \xi)$$

$k$ -th moments exist in general for  $k\xi < 1$ .

### Value

`rgpd` generates random deviates.

`dgpd` gives the density.

`pgpd` gives the distribution function.

`qgpd` gives the quantile function.

### References

[https://en.wikipedia.org/wiki/Generalized\\_Pareto\\_distribution](https://en.wikipedia.org/wiki/Generalized_Pareto_distribution)

### Examples

```
x <- rgpd(1000, u = 1, sigmau = 0.5, xi = 0.1)
xx <- seq(-1, 10, 0.01)
hist(x, breaks = 100, freq = FALSE, xlim = c(-1, 10))
lines(xx, dgpd(xx, u = 1, sigmau = 0.5, xi = 0.1))

plot(xx, dgpd(xx, u = 1, sigmau = 1, xi = 0), type = "l")
lines(xx, dgpd(xx, u = 0.5, sigmau = 1, xi = -0.3), col = "blue", lwd = 2)
lines(xx, dgpd(xx, u = 1.5, sigmau = 1, xi = 0.3), col = "red", lwd = 2)

plot(xx, dgpd(xx, u = 1, sigmau = 1, xi = 0), type = "l")
```

```
lines(xx, dgpd(xx, u = 1, sigma = 0.5, xi = 0), col = "blue", lwd = 2)
lines(xx, dgpd(xx, u = 1, sigma = 2, xi = 0), col = "red", lwd = 2)
```

---

integrate\_gk

*Adaptive Gauss-Kronrod Quadrature for multiple limits*


---

## Description

Integrates fun over the bounds [ lower, upper ] vectorized over lower and upper. Vectorized list structures of parameters can also be passed.

## Usage

```
integrate_gk(
  fun,
  lower,
  upper,
  params = list(),
  .tolerance = .Machine$double.eps^0.25,
  .max_iter = 100L
)
```

## Arguments

fun	A function to integrate. Must be vectorized and take one or two arguments, the first being points to evaluate at and the second (optionally) being parameters to apply. It must return a numeric vector the same length as its first input. Currently, infinite bounds are not supported.
lower, upper	Integration bounds. Must have the same length.
params	Parameters to pass as a second argument to fun. The actual parameters must have the same length as the number of integrals to compute. Can be a possibly nested list structures containing numeric vectors. Alternatively, can be a matrix with the same number of rows as the number of integrals to compute.
.tolerance	Absolute element-wise tolerance.
.max_iter	Maximum number of iterations. The number of integration intervals will be at most $\text{length}(\text{lower}) * \text{.max\_iter}$ . Therefore the maximum number of function evaluations per integration interval will be $15 * \text{.max\_iter}$ .

## Details

The integration error is estimated by the Gauss-Kronrod quadrature as the absolute difference between the 7-point quadrature and the 15-point quadrature. Integrals that did not converge will be bisected at the midpoint. The params object will be recursively subsetted on all numeric vectors with the same length as the number of observations.

**Value**

A vector of integrals with the  $i$ -th entry containing an approximation of  $\int_{\text{int\_lower}[i]}^{\text{upper}[i]} \text{fun}(t, \text{pick\_params\_at}(\text{params}, i)) dt$

**Examples**

```
# Argument recycling and parallel integration of two intervals
integrate_gk(sin, 0, c(pi, 2 * pi))

dist <- dist_exponential()
integrate_gk(
  function(x, p) dist$density(x, with_params = p),
  lower = 0, upper = 1:10,
  params = list(rate = 1 / 1:10)
)
dist$probability(1:10, with_params = list(rate = 1 / 1:10))
```

interval

*Intervals***Description**

Intervals

**Usage**

```
interval(
  range = c(-Inf, Inf),
  ...,
  include_lowest = closed,
  include_highest = closed,
  closed = FALSE,
  integer = FALSE,
  read_only = FALSE
)
```

is.Interval(x)

**Arguments**

range	The interval boundaries as a sorted two-element numeric vector.
...	First argument is used as the endpoint if range has length 1. Additional arguments, or any if range has length 2, cause a warning and will be ignored.
include_lowest	Is the lower boundary part of the interval?
include_highest	Is the upper boundary part of the interval?

closed	Is the interval closed?
integer	Is the interval only over the integers?
read_only	Make the interval object read-only?
x	An object.

**Value**

interval returns an Interval. is.Interval returns TRUE if x is an Interval, FALSE otherwise.

**See Also**

interval-operations

**Examples**

```
# The real line
interval()

# Closed unit interval
interval(c(0, 1), closed = TRUE)
# Alternative form
interval(0, 1, closed = TRUE)

# Non-negative real line
interval(c(0, Inf), include_lowest = TRUE)
```

---

interval-operations    *Convex union and intersection of intervals*

---

**Description**

Convex union and intersection of intervals

**Usage**

```
interval_union(..., intervals = list())

interval_intersection(..., intervals = list())
```

**Arguments**

...            appened to intervals if present.  
intervals      A list of Intervals.

**Value**

`interval_union` returns the convex union of all intervals in `intervals`. This is the smallest interval completely containing all intervals.

`interval_intersection` returns the set intersection of all intervals in `intervals`. The empty set is represented by the open interval  $(0, 0)$ .

**See Also**

`interval`

**Examples**

```
interval_union(  
  interval(c(0, 1), closed = TRUE),  
  interval(c(1, 2))  
)  
  
interval_union(  
  interval(c(0, 5)),  
  interval(c(1, 4), closed = TRUE)  
)  
  
# Convex union is not equal to set union:  
interval_union(  
  interval(c(0, 1)),  
  interval(c(2, 3))  
)  
  
# The empty union is {}  
interval_union()  
  
interval_intersection(  
  interval(c(0, 1)),  
  interval(c(0.5, 2))  
)  
  
interval_intersection(  
  interval(c(0, Inf)),  
  interval(c(-Inf, 0))  
)  
  
interval_intersection(  
  interval(c(0, Inf), include_lowest = TRUE),  
  interval(c(-Inf, 0), include_highest = TRUE)  
)  
  
interval_intersection(  
  interval(c(0, 5)),  
  interval(c(1, 6), closed = TRUE)  
)
```

```
# The empty intersection is (-Inf, Inf)
interval_intersection()
```

---

is.Distribution	<i>Test if object is a Distribution</i>
-----------------	---

---

### Description

Test if object is a Distribution

### Usage

```
is.Distribution(object)
```

### Arguments

object            An R object.

### Value

TRUE if object is a Distribution, FALSE otherwise.

### Examples

```
is.Distribution(dist_dirac())
```

---

k_matrix	<i>Cast to a TensorFlow matrix</i>
----------	------------------------------------

---

### Description

Cast to a TensorFlow matrix

### Usage

```
k_matrix(x, dtype = NULL)
```

### Arguments

x                    Numeric object to be converted to a matrix Tensor.  
 dtype                Type of the elements of the resulting tensor. Defaults to [k\\_floatx\(\)](#).

### Value

A two-dimensional `tf.Tensor` with values from `x`. The shape will be `(nrow(x), ncol(x))` where `x` is first converted to an R matrix via [as.matrix\(\)](#).

**Examples**

```

if (keras::is_keras_available()) {
  k_matrix(diag(1:3))
  k_matrix(diag(1:3), dtype = "int32")
  # Vectors are converted to columns:
  k_matrix(1:3)
}

```

Pareto

*The Pareto Distribution***Description**

These functions provide information about the Pareto distribution. `dpareto` gives the density, `ppareto` gives the distribution function, `qpareto` gives the quantile function and `rpareto` generates random deviates.

**Usage**

```
rpareto(n = 1L, shape = 0, scale = 1)
```

```
dpareto(x, shape = 1, scale = 1, log = FALSE)
```

```
ppareto(q, shape = 1, scale = 1, lower.tail = TRUE, log.p = FALSE)
```

```
qpareto(p, shape = 1, scale = 1, lower.tail = TRUE, log.p = FALSE)
```

**Arguments**

<code>n</code>	integer number of observations.
<code>shape</code>	shape parameter (must be positive).
<code>scale</code>	scale parameter (must be positive).
<code>x, q</code>	vector of quantiles.
<code>log, log.p</code>	logical; if TRUE, probabilities/densities $p$ are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P(X \leq x)$ , otherwise $P(X > x)$ .
<code>p</code>	vector of probabilities.

**Details**

If `shape` or `scale` are not specified, they assume the default values of 1.

The Pareto distribution with scale  $\theta$  and shape  $\xi$  has density

$$f(x) = \xi\theta^\xi / (x + \theta)^{\xi + 1}$$

The support is  $x \geq 0$ .

The Expected value exists if  $\xi > 1$  and is equal to

$$E(X) = \theta/(\xi - 1)$$

k-th moments exist in general for  $k < \xi$ .

### Value

rpareto generates random deviates.  
 dpareto gives the density.  
 ppareto gives the distribution function.  
 qpareto gives the quantile function.

### References

[https://en.wikipedia.org/wiki/Pareto\\_distribution](https://en.wikipedia.org/wiki/Pareto_distribution) - named Lomax therein.

### Examples

```
x <- rpareto(1000, shape = 10, scale = 5)
xx <- seq(-1, 10, 0.01)
hist(x, breaks = 100, freq = FALSE, xlim = c(-1, 10))
lines(xx, dpareto(xx, shape = 10, scale = 5))

plot(xx, dpareto(xx, shape = 10, scale = 5), type = "l")
lines(xx, dpareto(xx, shape = 3, scale = 5), col = "red", lwd = 2)

plot(xx, dpareto(xx, shape = 10, scale = 10), type = "l")
lines(xx, dpareto(xx, shape = 10, scale = 5), col = "blue", lwd = 2)
lines(xx, dpareto(xx, shape = 10, scale = 20), col = "red", lwd = 2)
```

---

plot\_distributions      *Plot several distributions*

---

### Description

Plot several distributions

### Usage

```
plot_distributions(
  ...,
  distributions = list(),
  .x,
  plots = c("density", "probability", "hazard"),
  with_params = list(),
  as_list = FALSE
)
```



**Arguments**

...	distribution objects (must be named)
distributions	Named list of distribution objects. This is concatenated with ...
.x	Numeric vector of points to evaluate at.
plots	Plots to be created. May be abbreviated. The plots will be stacked in the order given from top to bottom.
with_params	list of distribution parameters to be given to each distribution using with_params. If named, the names are matched to the distribution names. Otherwise, they are allocated positionally, index 1 corresponding to the first element of distributions, then all other elements from distributions followed by the arguments in ... in order.
as_list	return a list of ggplots instead of a patchwork?

**Value**

A stacked patchwork of the requested ggplots

**Examples**

```
rate <- 1
x <- rexp(20, rate)
d_emp <- dist_empirical(x, positive = TRUE)
d_exp <- dist_exponential()
plot_distributions(
  empirical = d_emp,
  theoretical = d_exp,
  estimated = d_exp,
  with_params = list(
    theoretical = list(rate = rate),
    estimated = list(rate = 1 / mean(x))
  ),
  .x = seq(1e-4, 5, length.out = 100)
)
```

---

predict.reservr\_keras\_model

*Predict individual distribution parameters*

---

**Description**

Predict individual distribution parameters

**Usage**

```
## S3 method for class 'reservr_keras_model'
predict(object, data, as_matrix = FALSE, ...)
```

**Arguments**

object	A compiled and trained <code>resrvr_keras_model</code> .
data	Input data compatible with the model.
as_matrix	Return a parameter matrix instead of a list structure?
...	ignored

**Value**

A parameter list suitable for the `with_params` argument of the distribution family used for the model. Contains one set of parameters per row in data.

**Examples**

```

if (keras::is_keras_available()) {
  dist <- dist_exponential()
  params <- list(rate = 1.0)
  N <- 100L
  rand_input <- runif(N)
  x <- dist$sample(N, with_params = params)

  tf_in <- keras::layer_input(1L)
  mod <- tf_compile_model(
    inputs = list(tf_in),
    intermediate_output = tf_in,
    dist = dist,
    optimizer = keras::optimizer_adam(),
    censoring = FALSE,
    truncation = FALSE
  )

  tf_fit <- fit(
    object = mod,
    x = k_matrix(rand_input),
    y = x,
    epochs = 10L,
    callbacks = list(
      callback_debug_dist_gradients(mod, k_matrix(rand_input), x)
    )
  )

  tf_preds <- predict(mod, data = k_matrix(rand_input))
}

```

**Description**

Determines the probability that claims occurring under a Poisson process with arrival intensity `expo` and reporting delay distribution `dist` during the time between `t_min` and `t_max` are reported between `tau_min` and `tau_max`.

**Usage**

```
prob_report(
  dist,
  intervals,
  expo = NULL,
  with_params = list(),
  .tolerance = .Machine$double.eps^0.5,
  .max_iter = 100L,
  .try_compile = TRUE
)
```

**Arguments**

<code>dist</code>	A reporting delay Distribution, or a compiled interval probability function.
<code>intervals</code>	A data frame with columns <code>xmin</code> , <code>xmax</code> , <code>tmin</code> , <code>tmax</code> . Claims occur within <code>[xmin, xmax]</code> and be reported within <code>[tmin, tmax]</code> .
<code>expo</code>	Poisson intensity. If given, must be a vectorised function that yields the intensity of the claim arrival process at a specified time. <code>expo = NULL</code> is equivalent to a constant intensity function. <code>expo</code> is only relevant up to a multiplicative constant.
<code>with_params</code>	Parameters of <code>dist</code> to use. Can be a parameter set with different values for each interval. If <code>dist</code> is a compiled interval probability function, <code>with_params</code> can be a matrix instead.
<code>.tolerance</code>	Absolute element-wise tolerance.
<code>.max_iter</code>	Maximum number of iterations. The number of integration intervals will be at most <code>length(lower) * .max_iter</code> . Therefor the maximum number of function evaluations per integration interval will be <code>15 * .max_iter</code> .
<code>.try_compile</code>	Try compiling the distributions probability function to speed up integration?

**Details**

The reporting probability is given by

$$\frac{P(x + d \text{ in } [tmin, tmax] \mid x \text{ in } [xmin, xmax])}{P(x \text{ in } [xmin, xmax])} = \frac{\int_{[xmin, xmax]} \text{expo}(x) P(x + d \text{ in } [tmin, tmax]) dx}{\int_{[xmin, xmax]} \text{expo}(x) dx}$$

`prob_report` uses `integrate_gk()` to compute the two integrals.

**Value**

A vector of reporting probabilities, with one entry per row of `intervals`.

**Examples**

```

dist <- dist_exponential()
ints <- data.frame(
  xmin = 0,
  xmax = 1,
  tmin = seq_len(10) - 1.0,
  tmax = seq_len(10)
)
params <- list(rate = rep(c(1, 0.5), each = 5))

prob_report(dist, ints, with_params = params)

```

---

quantile.Distribution *Quantiles of Distributions*

---

**Description**

Produces quantiles corresponding to the given probabilities with configurable distribution parameters.

**Usage**

```

## S3 method for class 'Distribution'
quantile(x, probs = seq(0, 1, 0.25), with_params = list(), ..., .start = 0)

```

**Arguments**

<code>x</code>	A Distribution.
<code>probs</code>	Quantiles to compute.
<code>with_params</code>	Optional list of distribution parameters. Note that if <code>x\$has_capability("quantile")</code> is false, <code>with_params</code> is assumed to contain only one set of parameters.
<code>...</code>	ignored
<code>.start</code>	Starting value if quantiles are computed numerically. Must be within the support of <code>x</code> .

**Details**

If `x$has_capability("quantile")` is true, this returns the same as `x$quantile(probs, with_params = with_params)`. In this case, `with_params` may contain separate sets of parameters for each quantile to be determined.

Otherwise, a numerical estimation of the quantiles is done using the density and probability function. This method assumes `with_params` to contain only one set of parameters. The strategy uses two steps:

1. Find the smallest and largest quantiles in `probs` using a newton method starting from `.start`.
2. Find the remaining quantiles with bisection using `stats::uniroot()`.

**Value**

The quantiles of  $x$  corresponding to probs with parameters with\_params.

**Examples**

```
# With quantiles available
dist <- dist_normal(sd = 1)
qq <- quantile(dist, probs = rep(0.5, 3), with_params = list(mean = 1:3))
stopifnot(all.equal(qq, 1:3))

# Without quantiles available
dist <- dist_ergangmix(shapes = list(1, 2, 3), scale = 1.0)
my_probs <- c(0, 0.01, 0.25, 0.5, 0.75, 1)
qq <- quantile(
  dist, probs = my_probs,
  with_params = list(probs = list(0.5, 0.3, 0.2)), .start = 2
)

all.equal(dist$probability(qq, with_params = list(probs = list(0.5, 0.3, 0.2))), my_probs)
# Careful: Numerical estimation of extreme quantiles can result in out-of-bounds values.
# The correct 0-quantile would be 0 in this case, but it was estimated < 0.
qq[1L]
```

---

 softmax

*Soft-Max function*


---

**Description**

Softmax for a vector  $x$  is defined as

**Usage**

```
softmax(x)
```

```
dsoftmax(x)
```

**Arguments**

$x$                     A numeric vector or matrix

**Details**

$$s_i = \exp(x_i) / \sum_k \exp(x_k)$$

It satisfies  $\text{sum}(s) == 1.0$  and can be used to smoothly enforce a sum constraint.

**Value**

softmax returns the softmax of  $x$ ; rowwise if  $x$  is a matrix.

dsoftmax returns the Jacobi-matrix of softmax( $x$ ) at  $x$ .  $x$  must be a vector.

**Examples**

```
softmax(c(5, 5))
softmax(diag(nrow = 5, ncol = 6))
```

---

tf_compile_model	<i>Compile a Keras model for truncated data under dist</i>
------------------	--

---

**Description**

Compile a Keras model for truncated data under dist

**Usage**

```
tf_compile_model(
  inputs,
  intermediate_output,
  dist,
  optimizer,
  censoring = TRUE,
  truncation = TRUE,
  metrics = NULL,
  sample_weight_mode = NULL,
  weighted_metrics = NULL,
  target_tensors = NULL
)
```

**Arguments**

inputs	List of keras input layers
intermediate_output	Intermediate model layer to be used as input to distribution parameters
dist	A Distribution to use for compiling the loss and parameter outputs
optimizer	String (name of optimizer) or optimizer instance. For most models, this defaults to "rmsprop"
censoring	A flag, whether the compiled model should support censored observations. Set to FALSE for higher efficiency. <code>fit(...)</code> will error if the resulting model is used to fit censored observations.
truncation	A flag, whether the compiled model should support truncated observations. Set to FALSE for higher efficiency. <code>fit(...)</code> will warn if the resulting model is used to fit truncated observations.

- metrics** List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a `keras$metrics$Metric` class instance. See `?tf$keras$metrics`. Typically you will use `metrics=list('accuracy')`. A function is any callable with the signature `result = fn(y_true, y_pred)`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics=list(output_a = 'accuracy', output_b = c('accuracy', 'mse'))`. You can also pass a list to specify a metric or a list of metrics for each output, such as `metrics=list(list('accuracy'), list('accuracy', 'mse'))` or `metrics=list('accuracy', c('accuracy', 'mse'))`. When you pass the strings 'accuracy' or 'acc', this is converted to one of `tf.keras.metrics.BinaryAccuracy`, `tf.keras.metrics.CategoricalAccuracy`, `tf.keras.metrics.SparseCategoricalAccuracy` based on the loss function used and the model output shape. A similar conversion is done for the strings 'crossentropy' and 'ce'.
- sample\_weight\_mode** If you need to do timestep-wise sample weighting (2D weights), set this to "temporal". NULL defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a list of modes.
- weighted\_metrics** List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- target\_tensors** By default, Keras will create a placeholder for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensor (in turn, Keras will not expect external data for these targets at training time), you can specify them via the `target_tensors` argument. It should be a single tensor (for a single-output sequential model).

## Value

A `reservr_keras_model` that can be used to train truncated and censored observations from `dist` based on input data from `inputs`.

## Examples

```
dist <- dist_exponential()
params <- list(rate = 1.0)
N <- 100L
rand_input <- runif(N)
x <- dist$sample(N, with_params = params)

if (keras::is_keras_available()) {
  tf_in <- keras::layer_input(1L)
  mod <- tf_compile_model(
    inputs = list(tf_in),
    intermediate_output = tf_in,
    dist = dist,
    optimizer = keras::optimizer_adam(),
    censoring = FALSE,
    truncation = FALSE
  )
}
```

```

  )
}
```

---

tf\_initialise\_model    *Initialise model weights to a global parameter fit*

---

### Description

Initialises a compiled `reservr_keras_model` weights such that the predictions are equal to, or close to, the distribution parameters given by `params`.

### Usage

```

tf_initialise_model(
  model,
  params,
  mode = c("scale", "perturb", "zero", "none")
)
```

### Arguments

<code>model</code>	A <code>reservr_compiled_model</code> obtained by <code>tf_compile_model()</code> .
<code>params</code>	A list of distribution parameters compatible with <code>model</code> .
<code>mode</code>	An initialisation mode
	<b>scale</b> Initialise the biases according to <code>params</code> and the kernels uniform on $[-0.1, 0.1] * \text{bias scale}$ .
	<b>perturb</b> Initialise the biases according to <code>params</code> and leave the kernels as is.
	<b>zero</b> Initialise the biases according to <code>params</code> and set the kernel to zero.
	<b>none</b> Don't modify the weights.

### Value

Invisibly `model` with changed weights

### Examples

```

dist <- dist_exponential()
group <- sample(c(0, 1), size = 100, replace = TRUE)
x <- dist$sample(100, with_params = list(rate = group + 1))
global_fit <- fit(dist, x)

if (keras::is_keras_available()) {
  library(keras)
  l_in <- layer_input(shape = 1L)
  mod <- tf_compile_model(
    inputs = list(l_in),
```



```

    intermediate_output = l_in,
    dist = dist,
    optimizer = optimizer_adam(),
    censoring = FALSE,
    truncation = FALSE
  )
  tf_initialise_model(mod, global_fit$params)
  fit_history <- fit(
    mod,
    x = group,
    y = x,
    epochs = 200L
  )

  predicted_means <- predict(mod, data = k_constant(c(0, 1)))
}

```

---

truncate\_claims

*Truncate claims data subject to reporting delay*


---

## Description

Truncate claims data subject to reporting delay

## Usage

```
truncate_claims(data, accident, delay, time, .report_col = "report")
```

## Arguments

data	Full claims data including IBNR
accident	Accident times. May be an unquoted column name from data.
delay	Reporting delays. May be an unquoted column name from data.
time	Observation time (scalar number or one per claim). Claims with <code>accident + delay &gt; time</code> will be truncated. Set <code>time = Inf</code> to only compute reporting times and perform no truncation.
.report_col	NULL or a column name to store the reporting time <code>report = accident + delay</code> .

## Value

Truncated data. The reporting time is stored in a column named by `.report_col` unless `.report_col` is NULL. If both `.report_col` is NULL and `time` contains only Infs, a warning will be issued since data will be returned unchanged and no work will be done.

**Examples**

```
claims_full <- data.frame(
  acc = runif(100),
  repdel = rexp(100)
)
tau <- 2.0
truncate_claims(claims_full, acc, repdel, tau)
```

trunc\_obs

*Define a set of truncated observations***Description**

If  $x$  is missing, both  $x_{\min}$  and  $x_{\max}$  must be specified.

**Usage**

```
trunc_obs(x, xmin = x, xmax = x, tmin = -Inf, tmax = Inf, w = 1)

as_trunc_obs(.data)

truncate_obs(.data, tmin_new = -Inf, tmax_new = Inf, .partial = FALSE)

repdel_obs(.data, accident, delay, time, .truncate = FALSE)
```

**Arguments**

<code>x</code>	Observations
<code>xmin, xmax</code>	Censoring bounds. If <code>xmin != xmax</code> , <code>x</code> must be NA.
<code>tmin, tmax</code>	Truncation bounds. May vary per observation.
<code>w</code>	Case weights
<code>.data</code>	A data frame or numeric vector.
<code>tmin_new</code>	New truncation minimum
<code>tmax_new</code>	New truncation maximum
<code>.partial</code>	Enable partial truncation of censored observations? This could potentially create inconsistent data if the actual observation lies outside of the truncation bounds but the censoring interval overlaps.
<code>accident</code>	accident time (unquoted, evaluated in <code>.data</code> )
<code>delay</code>	reporting delay (unquoted, evaluated in <code>.data</code> )
<code>time</code>	evaluation time (unquoted, evaluated in <code>.data</code> )
<code>.truncate</code>	Should claims reported after <code>time</code> be silently discarded? If there are claims reported after <code>time</code> and <code>.truncate</code> is FALSE, an error will be raised.

## Details

Uncensored observations must satisfy  $t_{\min} \leq x_{\min} = x = x_{\max} \leq t_{\max}$ . Censored observations must satisfy  $t_{\min} \leq x_{\min} < x_{\max} \leq t_{\max}$  and  $x = \text{NA}$ .

## Value

**trunc\_obs**: A trunc\_obs tibble with columns x, xmin, xmax, tmin and tmax describing possibly interval-censored observations with truncation

as\_trunc\_obs returns a trunc\_obs tibble.

truncate\_obs returns a trunc\_obs tibble with possibly fewer observations than .data and updated truncation bounds.

repdel\_obs returns a trunc\_obs tibble corresponding to the reporting delay observations of each claim. If .truncate is FALSE, the result is guaranteed to have the same number of rows as .data.

## Examples

```
N <- 100
x <- rexp(N, 0.5)

# Random, observation dependent truncation intervals
tmin <- runif(N, 0, 1)
tmax <- tmin + runif(N, 1, 2)

oob <- x < tmin | x > tmax
x <- x[!oob]
tmin <- tmin[!oob]
tmax <- tmax[!oob]

# Number of observations after truncation
N <- length(x)

# Randomly interval censor 30% of observations
cens <- rbinom(N, 1, 0.3) == 1L
xmin <- x
xmax <- x
xmin[cens] <- pmax(tmin[cens], floor(x[cens]))
xmax[cens] <- pmin(tmax[cens], ceiling(x[cens]))
x[cens] <- NA

trunc_obs(x, xmin, xmax, tmin, tmax)

as_trunc_obs(c(1, 2, 3))
as_trunc_obs(data.frame(x = 1:3, tmin = 0, tmax = 10))
as_trunc_obs(data.frame(x = c(1, NA), xmin = c(1, 2), xmax = c(1, 3)))
truncate_obs(1:10, tmin_new = 2.0, tmax_new = 8.0)
```

---

weighted_moments	<i>Compute weighted moments</i>
------------------	---------------------------------

---

**Description**

Compute weighted moments

**Usage**

```
weighted_moments(x, w, n = 2L, center = TRUE)
```

**Arguments**

x	Observations
w	Case weights (optional)
n	Number of moments to calculate
center	Calculate centralized moments (default) or noncentralized moments, i.e. $E((X - E(X))^k)$ or $E(X^k)$ .

**Value**

A vector of length n where the kth entry is the kth weighted moment of x with weights w. If center is TRUE the moments are centralized, i.e.  $E((X - E(X))^k)$ . The first moment is never centralized. The moments are scaled with  $1 / \text{sum}(w)$ , so they are not de-biased.

e.g. the second central weighted moment `weighted_moment(x, w)[2L]` is equal to `var(rep(x, w)) * (sum(w) - 1) / sum(w)` for integer w

**See Also**

Other weighted statistics: [weighted\\_quantile\(\)](#), [weighted\\_tabulate\(\)](#)

**Examples**

```
weighted_moments(rexp(100))
weighted_moments(c(1, 2, 3), c(1, 2, 3))
c(mean(rep(1:3, 1:3)), var(rep(1:3, 1:3)) * 5 / 6)
```

---

weighted_quantile	<i>Compute weighted quantiles</i>
-------------------	-----------------------------------

---

**Description**

Compute weighted quantiles

**Usage**

```
weighted_quantile(x, w, probs)
```

```
weighted_median(x, w)
```

**Arguments**

x	Observations
w	Case weights (optional)
probs	Quantiles to calculate

**Value**

A vector the same length as probs with the corresponding weighted quantiles of x with weight w. For integer weights, this is equivalent to `quantile(rep(x, w), probs)`

The weighted median of x with weights w. For integer weights, this is equivalent to `median(rep(x, w))`

**See Also**

Other weighted statistics: [weighted\\_moments\(\)](#), [weighted\\_tabulate\(\)](#)

**Examples**

```
weighted_median(1:6)
weighted_median(1:3, c(1, 4, 9))
weighted_median(1:3, c(9, 4, 1))

weighted_quantile(1:3, c(1, 4, 9), seq(0.0, 1.0, by = 0.25))
quantile(rep(1:3, c(1, 4, 9)), seq(0.0, 1.0, by = 0.25))
```

---

weighted\_tabulate      *Compute weighted tabulations*

---

**Description**

Computes the sum of `w` grouped by `bin`. If `w` is missing the result is equivalent to `tabulate(bin, nbins)`

**Usage**

```
weighted_tabulate(bin, w, nbins = max(1L, bin, na.rm = TRUE))
```

**Arguments**

<code>bin</code>	An integer vector with values from 1L to <code>nbins</code>
<code>w</code>	Weights per entry in <code>bin</code> .
<code>nbins</code>	Number of bins

**Value**

A vector with length `nbins` where the `i`th result is equal to `sum(w[bin == i])` or `sum(bin == i)` if `w` is missing. For integer weights, this is equivalent to `tabulate(rep(bin, w), nbins)`.

**See Also**

Other weighted statistics: `weighted_moments()`, `weighted_quantile()`

**Examples**

```
weighted_tabulate(c(1, 1, 2))  
weighted_tabulate(c(1, 1, 2), nbins = 3L)  
weighted_tabulate(c(1, 1, 2), w = c(0.5, 0.5, 1), nbins = 3L)
```

# Index

## \* Distributions

- dist\_bdegp, 23
- dist\_beta, 24
- dist\_binomial, 25
- dist\_blended, 26
- dist\_dirac, 27
- dist\_discrete, 28
- dist\_empirical, 29
- dist\_erlangmix, 31
- dist\_exponential, 32
- dist\_gamma, 33
- dist\_genpareto, 34
- dist\_lognormal, 35
- dist\_mixture, 36
- dist\_negbinomial, 37
- dist\_normal, 38
- dist\_pareto, 39
- dist\_poisson, 40
- dist\_translate, 41
- dist\_trunc, 42
- dist\_uniform, 43
- dist\_weibull, 44
- Distribution, 9

## \* distribution fitting functions

- fit\_blended, 48
- fit\_dist, 49
- fit\_erlang\_mixture, 52
- fit\_mixture, 53

## \* weighted statistics

- weighted\_moments, 76
- weighted\_quantile, 77
- weighted\_tabulate, 78

as.matrix(), 62

as\_params, 3

as\_trunc\_obs(trunc\_obs), 74

as\_trunc\_obs(), 46, 48, 50, 52, 54

blended\_transition, 4

blended\_transition\_inv  
(blended\_transition), 4

callback\_adaptive\_lr, 6

callback\_debug\_dist\_gradients, 8

callback\_reduce\_lr\_on\_plateau(), 7

compile(), 46

density(), 30

dgpd (GenPareto), 56

dist\_bdegp, 19, 23, 25–35, 37–44

dist\_beta, 19, 23, 24, 26–35, 37–44

dist\_binomial, 19, 23, 25, 25, 27–35, 37–44

dist\_blended, 19, 23, 25, 26, 26, 28–35, 37–44

dist\_dirac, 19, 23, 25–27, 27, 29–35, 37–44

dist\_dirac(), 54

dist\_discrete, 19, 23, 25–28, 28, 30–35, 37–44

dist\_empirical, 19, 23, 25–29, 29, 31–35, 37–44

dist\_erlangmix, 19, 23, 25–30, 31, 32–35, 37–44

dist\_exponential, 19, 23, 25–31, 32, 33–35, 37–44

dist\_gamma, 19, 23, 25–32, 33, 34, 35, 37–44

dist\_genpareto, 19, 23, 25–33, 34, 35, 37–44

dist\_genpareto1 (dist\_genpareto), 34

dist\_lognormal, 19, 23, 25–34, 35, 37–44

dist\_mixture, 19, 23, 25–35, 36, 38–44

dist\_negbinomial, 19, 23, 25–35, 37, 37, 39–44

dist\_normal, 19, 23, 25–35, 37, 38, 38, 40–44

dist\_pareto, 19, 23, 25–35, 37–39, 39, 41–44

dist\_poisson, 19, 23, 25–35, 37–40, 40, 42–44

dist\_translate, 19, 23, 25–35, 37–41, 41, 42–44

dist\_trunc, 19, 23, 25–35, 37–42, 42, 43, 44

dist\_uniform, 19, 23, 25–35, 37–42, 43, 44

- dist\_weibull, [19](#), [23](#), [25–35](#), [37–43](#), [44](#)
- Distribution, [9](#), [23](#), [25–35](#), [37–44](#)
- dpareto (Pareto), [63](#)
- dsoftmax (softmax), [69](#)
- evmix::gpd, [34](#)
- fit.Distribution (fit\_dist), [49](#)
- fit.reservr\_keras\_model, [45](#)
- fit\_blended, [48](#), [50](#), [53](#), [54](#)
- fit\_dist, [49](#), [49](#), [53](#), [54](#)
- fit\_dist(), [51](#)
- fit\_dist\_direct (fit\_dist), [49](#)
- fit\_dist\_start
  - (fit\_dist\_start.MixtureDistribution), [51](#)
- fit\_dist\_start(), [48](#), [50–52](#), [54](#)
- fit\_dist\_start.MixtureDistribution, [51](#)
- fit\_erlang\_mixture, [49](#), [50](#), [52](#), [54](#)
- fit\_mixture, [49](#), [50](#), [53](#), [53](#)
- flatten\_bounds (flatten\_params), [55](#)
- flatten\_params, [55](#)
- flatten\_params\_matrix (flatten\_params), [55](#)
- GenPareto, [56](#)
- inflate\_params (flatten\_params), [55](#)
- integrate\_gk, [58](#)
- integrate\_gk(), [67](#)
- interval, [59](#)
- interval-operations, [60](#)
- interval\_intersection
  - (interval-operations), [60](#)
- interval\_union (interval-operations), [60](#)
- is.Distribution, [62](#)
- is.Interval (interval), [59](#)
- k\_floatx(), [62](#)
- k\_matrix, [62](#)
- keras::callback\_reduce\_lr\_on\_plateau(), [6](#)
- keras::fit(), [7](#), [8](#)
- keras::fit.keras.engine.training.Model(), [45](#)
- logKDE::logdensity\_fft, [30](#)
- logKDE::logdensity\_fft(), [30](#)
- Pareto, [39](#), [63](#)
- pgpd (GenPareto), [56](#)
- plot\_distributions, [64](#)
- ppareto (Pareto), [63](#)
- predict.reservr\_keras\_model, [65](#)
- prob\_report, [66](#)
- qgpd (GenPareto), [56](#)
- qpareto (Pareto), [63](#)
- quantile.Distribution, [68](#)
- repedl\_obs (trunc\_obs), [74](#)
- rgpd (GenPareto), [56](#)
- rpareto (Pareto), [63](#)
- softmax, [69](#)
- stats::Beta, [24](#)
- stats::Binomial, [25](#)
- stats::density, [30](#)
- stats::ecdf, [30](#)
- stats::Exponential, [32](#)
- stats::GammaDist, [33](#)
- stats::Lognormal, [35](#)
- stats::NegBinomial, [37](#)
- stats::Normal, [38](#)
- stats::Poisson, [40](#)
- stats::quantile, [30](#)
- stats::Uniform, [43](#)
- stats::uniroot(), [68](#)
- stats::Weibull, [44](#)
- tf\_compile\_model, [70](#)
- tf\_compile\_model(), [8](#), [17](#), [45](#), [72](#)
- tf\_initialise\_model, [72](#)
- trunc\_obs, [74](#)
- trunc\_obs(), [48](#), [50](#), [52](#), [54](#)
- truncate\_claims, [73](#)
- truncate\_obs (trunc\_obs), [74](#)
- weighted\_median (weighted\_quantile), [77](#)
- weighted\_moments, [76](#), [77](#), [78](#)
- weighted\_quantile, [76](#), [77](#), [78](#)
- weighted\_tabulate, [76](#), [77](#), [78](#)