Package ‘greta’

August 9, 2019

Type Package

Title Simple and Scalable Statistical Modelling in R

Version 0.3.1

Date 2019-08-08

Description Write statistical models in R and fit them by MCMC and optimisation on CPUs and GPUs, using Google 'TensorFlow'.
greta lets you write your own model like in BUGS, JAGS and Stan, except that you write models right in R, it scales well to massive datasets, and it's easy to extend and build on.
See the website for more information, including tutorials, examples, package documentation, and the greta forum.

License Apache License 2.0

URL https://greta-stats.org

BugReports https://github.com/greta-dev/greta/issues

SystemRequirements Python (>= 2.7.0) with header files and shared library; TensorFlow (v1.14; https://www.tensorflow.org/); TensorFlow Probability (v0.7.0; https://www.tensorflow.org/probability/)

Encoding UTF-8

LazyData true

Depends R (>= 3.0)

Collate 'package.R' 'utils.R' 'tf_functions.R' 'overloaded.R' 'node_class.R' 'node_types.R' 'variable.R' 'probability_distributions.R' 'mixture.R' 'joint.R' 'unknowns_class.R' 'greta_array_class.R' 'as_data.R' 'distribution.R' 'operators.R' 'functions.R' 'transforms.R' 'structures.R' 'extract_replace_combine.R' 'dag_class.R' 'greta_model_class.R' 'progress_bar.R' 'inference_class.R' 'samplers.R' 'optimisers.R' 'inference.R' 'install_tensorflow.R' 'internals.R' 'calculate.R' 'callbacks.R'

Imports R6, tensorflow (>= 1.13.0), reticulate, progress (>= 1.2.0), future, coda, methods

Suggests knitr, rmarkdown, DiagrammeR, bayesplot, lattice, testthat, mvtnorm, MCMCpack, rmutil, extraDistr, truncdist, tidyverse, fields, MASS, abind, spelling

VignetteBuilder knitr

RoxygenNote 6.1.1

Language en-GB

NeedsCompilation no

Author Nick Golding [aut, cre] (<https://orcid.org/0000-0001-8916-5570>),
Simon Dirmeier [ctb],
Adam Fleischhacker [ctb],
Shirin Glander [ctb],
Martin Ingram [ctb],
Lee Hazel [ctb],
Tiphaine Martin [ctb],
Matt Mulvahill [ctb],
Michael Quinn [ctb],
David Smith [ctb],
Paul Teetor [ctb],
Jian Yen [ctb]

Maintainer Nick Golding <`nick.golding.research@gmail.com`>

Repository CRAN

Date/Publication 2019-08-09 04:40:03 UTC

R topics documented:

as_data — *convert other objects to greta arrays*

Description

define an object in an R session as a data greta array for use as data in a greta model.

Usage

```
as_data(x)
```

Arguments

x	an R object that can be coerced to a greta_array (see details).

Details

as_data() can currently convert R objects to greta_arrays if they are numeric or logical vectors, matrices or arrays; or if they are dataframes with only numeric (including integer) or logical elements. Logical elements are always converted to numerics. R objects cannot be converted if they contain missing (NA) or infinite (-Inf or Inf) values.

Examples

```
## Not run:

# numeric/integer/logical vectors, matrices and arrays can all be coerced to
# data greta arrays

vec <- rnorm(10)
mat <- matrix(seq_len(3 * 4), nrow = 3)
arr <- array(sample(c(TRUE, FALSE), 2 * 2 * 2, replace = TRUE),
             dim = c(2, 2, 2))
(a <- as_data(vec))
(b <- as_data(mat))
(c <- as_data(arr))

# dataframes can also be coerced, provided all the columns are numeric,
# integer or logical
df <- data.frame(x1 = rnorm(10),
                 x2 = sample(1L:10L),
                 x3 = sample(c(TRUE, FALSE), 10, replace = TRUE))
(d <- as_data(df))

## End(Not run)
```

calculate *calculate greta arrays given fixed values*

Description

Calculate the values that greta arrays would take, given temporary values for the greta arrays on which they depend, and return them as numeric R arrays. This can be used to check the behaviour of your model or make predictions to new data after model fitting.

Usage

```
calculate(target, values = list(), precision = c("double", "single"))
```

Arguments

`target`	a greta array for which to calculate the value
`values`	a named list giving temporary values of the greta arrays with which `target` is connected, or an `mcmc.list` object returned by `mcmc`.
`precision`	the floating point precision to use when calculating values.

Details

The greta arrays named in `values` need not be variables, they can also be other operations or even data.

At present, if `values` is a named list it must contain values for *all* of the variable greta arrays with which `target` is connected, even values are given for intermediate operations, or the target doesn't depend on the variable. That may be relaxed in a future release.

Value

A numeric R array with the same dimensions as `target`, giving the values it would take conditioned on the fixed values given by `values`.

Examples

```
## Not run:

# define a variable greta array, and another that is calculated from it
# then calculate what value y would take for different values of x
x <- normal(0, 1, dim = 3)
a <- lognormal(0, 1)
y <- sum(x ^ 2) + a
calculate(y, list(x = c(0.1, 0.2, 0.3), a = 2))

# if the greta array only depends on data,
# you can pass an empty list to values (this is the default)
x <- ones(3, 3)
y <- sum(x)
```

```
calculate(y)

# define a model
alpha <- normal(0, 1)
beta <- normal(0, 1)
sigma <- lognormal(1, 0.1)
mu <- alpha + iris$Petal.Length * beta
distribution(iris$Petal.Width) <- normal(mu, sigma)
m <- model(alpha, beta, sigma)

# calculate intermediate greta arrays, given some parameter values
calculate(mu[1:5], list(alpha = 1, beta = 2, sigma = 0.5))
calculate(mu[1:5], list(alpha = -1, beta = 0.2, sigma = 0.5))


# fit the model then calculate samples at a new greta array
draws <- mcmc(m, n_samples = 500)
petal_length_plot <- seq(min(iris$Petal.Length),
                         max(iris$Petal.Length),
                         length.out = 100)
mu_plot <- alpha + petal_length_plot * beta
mu_plot_draws <- calculate(mu_plot, draws)

# plot the draws
mu_est <- colMeans(mu_plot_draws[[1]])
plot(mu_est ~ petal_length_plot, type = "n",
     ylim = range(mu_plot_draws[[1]]))
apply(mu_plot_draws[[1]], 1, lines,
      x = petal_length_plot, col = grey(0.8))
lines(mu_est ~ petal_length_plot, lwd = 2)

## End(Not run)
```

`distribution` *define a distribution over data*

Description

`distribution` defines probability distributions over observed data, e.g. to set a model likelihood.

Usage

```
distribution(greta_array) <- value

distribution(greta_array)
```

Arguments

`greta_array`	a data greta array. For the assignment method it must not already have a probability distribution assigned
`value`	a greta array with a distribution (see `distributions`)

Details

The extract method returns the greta array if it has a distribution, or `NULL` if it doesn't. It has no real use-case, but is included for completeness

Examples

```
## Not run:

# define a model likelihood

# observed data and mean parameter to be estimated
# (explicitly coerce data to a greta array so we can refer to it later)
y <- as_data(rnorm(5, 0, 3))

mu <- uniform(-3, 3)

# define the distribution over y (the model likelihood)
distribution(y) <- normal(mu, 1)

# get the distribution over y
distribution(y)

## End(Not run)
```

`distributions` *probability distributions*

Description

These functions can be used to define random variables in a greta model. They return a variable greta array that follows the specified distribution. This variable greta array can be used to represent a parameter with prior distribution, combined into a mixture distribution using `mixture`, or used with `distribution` to define a distribution over a data greta array.

Usage

```
uniform(min, max, dim = NULL)

normal(mean, sd, dim = NULL, truncation = c(-Inf, Inf))

lognormal(meanlog, sdlog, dim = NULL, truncation = c(0, Inf))
```

```
bernoulli(prob, dim = NULL)

binomial(size, prob, dim = NULL)

beta_binomial(size, alpha, beta, dim = NULL)

negative_binomial(size, prob, dim = NULL)

hypergeometric(m, n, k, dim = NULL)

poisson(lambda, dim = NULL)

gamma(shape, rate, dim = NULL, truncation = c(0, Inf))

inverse_gamma(alpha, beta, dim = NULL, truncation = c(0, Inf))

weibull(shape, scale, dim = NULL, truncation = c(0, Inf))

exponential(rate, dim = NULL, truncation = c(0, Inf))

pareto(a, b, dim = NULL, truncation = c(0, Inf))

student(df, mu, sigma, dim = NULL, truncation = c(-Inf, Inf))

laplace(mu, sigma, dim = NULL, truncation = c(-Inf, Inf))

beta(shape1, shape2, dim = NULL, truncation = c(0, 1))

cauchy(location, scale, dim = NULL, truncation = c(-Inf, Inf))

chi_squared(df, dim = NULL, truncation = c(0, Inf))

logistic(location, scale, dim = NULL, truncation = c(-Inf, Inf))

f(df1, df2, dim = NULL, truncation = c(0, Inf))

multivariate_normal(mean, Sigma, n_realisations = NULL,
  dimension = NULL)

wishart(df, Sigma)

lkj_correlation(eta, dimension = 2)

multinomial(size, prob, n_realisations = NULL, dimension = NULL)

categorical(prob, n_realisations = NULL, dimension = NULL)

dirichlet(alpha, n_realisations = NULL, dimension = NULL)
```

```
dirichlet_multinomial(size, alpha, n_realisations = NULL,
  dimension = NULL)
```

Arguments

`min, max`	scalar values giving optional limits to `uniform` variables. Like `lower` and `upper`, these must be specified as numerics, they cannot be greta arrays (though see details for a workaround). Unlike `lower` and `upper`, they must be finite. `min` must always be less than `max`.
`dim`	the dimensions of the greta array to be returned, either a scalar or a vector of positive integers. See details.
`mean, meanlog, location, mu`	unconstrained parameters
`sd, sdlog, sigma, lambda, shape, rate, df, scale, shape1, shape2, alpha, beta, df1, df2, a, b, eta`	positive parameters, `alpha` must be a vector for `dirichlet` and `dirichlet_multinomial`.
`truncation`	a length-two vector giving values between which to truncate the distribution, similarly to the `lower` and `upper` arguments to `variable`
`prob`	probability parameter (`0 < prob < 1`), must be a vector for `multinomial` and `categorical`
`size, m, n, k`	positive integer parameter
`Sigma`	positive definite variance-covariance matrix parameter
`n_realisations`	the number of independent realisation of a multivariate distribution
`dimension`	the dimension of a multivariate distribution

Details

The discrete probability distributions (`bernoulli`, `binomial`, `negative_binomial`, `poisson`, `multinomial`, `categorical`, `dirichlet_multinomial`) can be used when they have fixed values (e.g. defined as a likelihood using `distribution`, but not as unknown variables.

For univariate distributions `dim` gives the dimensions of the greta array to create. Each element of the greta array will be (independently) distributed according to the distribution. `dim` can also be left at its default of `NULL`, in which case the dimension will be detected from the dimensions of the parameters (provided they are compatible with one another).

For multivariate distributions (`multivariate_normal()`, `multinomial()`, `categorical()`, `dirichlet()`, and `dirichlet_multinomial()`) each row of the output and parameters corresponds to an independent realisation. If a single realisation or parameter value is specified, it must therefore be a row vector (see example). `n_realisations` gives the number of rows/realisations, and `dimension` gives the dimension of the distribution. Ie. a bivariate normal distribution would be produced with `multivariate_normal(...,dimension = 2)`. The dimension can usually be detected from the parameters.

`multinomial()` does not check that observed values sum to `size`, and `categorical()` does not check that only one of the observed entries is 1. It's the user's responsibility to check their data matches the distribution!

The parameters of `uniform` must be fixed, not greta arrays. This ensures these values can always be transformed to a continuous scale to run the samplers efficiently. However, a hierarchical `uniform` parameter can always be created by defining a `uniform` variable constrained between 0 and 1, and then transforming it to the required scale. See below for an example.

Wherever possible, the parameterisations and argument names of greta distributions match commonly used R functions for distributions, such as those in the `stats` or `extraDistr` packages. The following table states the distribution function to which greta's implementation corresponds:

greta	reference
`uniform`	stats::dunif
`normal`	stats::dnorm
`lognormal`	stats::dlnorm
`bernoulli`	extraDistr::dbern
`binomial`	stats::dbinom
`beta_binomial`	extraDistr::dbbinom
`negative_binomial`	stats::dnbinom
`hypergeometric`	stats::dhyper
`poisson`	stats::dpois
`gamma`	stats::dgamma
`inverse_gamma`	extraDistr::dinvgamma
`weibull`	stats::dweibull
`exponential`	stats::dexp
`pareto`	extraDistr::dpareto
`student`	extraDistr::dlst
`laplace`	extraDistr::dlaplace
`beta`	stats::dbeta
`cauchy`	stats::dcauchy
`chi_squared`	stats::dchisq
`logistic`	stats::dlogis
`f`	stats::df
`multivariate_normal`	mvtnorm::dmvnorm
`multinomial`	stats::dmultinom
`categorical`	stats::dmultinom (size = 1)
`dirichlet`	extraDistr::ddirichlet
`dirichlet_multinomial`	extraDistr::ddirmnom
`wishart`	stats::rWishart
`lkj_correlation`	rethinking::dlkjcorr

Examples

```
## Not run:

# a uniform parameter constrained to be between 0 and 1
phi <- uniform(min = 0, max = 1)

# a length-three variable, with each element following a standard normal
# distribution
alpha <- normal(0, 1, dim = 3)

# a length-three variable of lognormals
sigma <- lognormal(0, 3, dim = 3)
```

```
# a hierarchical uniform, constrained between alpha and alpha + sigma,
eta <- alpha + uniform(0, 1, dim = 3) * sigma

# a hierarchical distribution
mu <- normal(0, 1)
sigma <- lognormal(0, 1)
theta <- normal(mu, sigma)

# a vector of 3 variables drawn from the same hierarchical distribution
thetas <- normal(mu, sigma, dim = 3)

# a matrix of 12 variables drawn from the same hierarchical distribution
thetas <- normal(mu, sigma, dim = c(3, 4))

# a multivariate normal variable, with correlation between two elements
# note that the parameter must be a row vector
Sig <- diag(4)
Sig[3, 4] <- Sig[4, 3] <- 0.6
theta <- multivariate_normal(t(rep(mu, 4)), Sig)

# 10 independent replicates of that
theta <- multivariate_normal(t(rep(mu, 4)), Sig, n_realisations = 10)

# 10 multivariate normal replicates, each with a different mean vector,
# but the same covariance matrix
means <- matrix(rnorm(40), 10, 4)
theta <- multivariate_normal(means, Sig, n_realisations = 10)
dim(theta)

# a Wishart variable with the same covariance parameter
theta <- wishart(df = 5, Sigma = Sig)


## End(Not run)
```

---

extract-replace-combine

extract, replace and combine greta arrays

Description

Generic methods to extract and replace elements of greta arrays, or to combine greta arrays.

Arguments

`x`	a greta array
`i, j`	indices specifying elements to extract or replace
`n`	a single integer, as in `utils::head()` and `utils::tail()`

`nrow, ncol`	optional dimensions for the resulting greta array when x is not a matrix.
`value`	for `` `[<-` `` a greta array to replace elements, for `` `dim<-` `` either NULL or a numeric vector of dimensions
`...`	either further indices specifying elements to extract or replace (`[`), or multiple greta arrays to combine (`cbind()`, `rbind()` & `c()`), or additional arguments (`rep()`, `head()`, `tail()`)
`drop, recursive`	generic arguments that are ignored for greta arrays

Details

`diag()` can be used to extract or replace the diagonal part of a square and two-dimensional greta array, but it cannot be used to create a matrix-like greta array from a scalar or vector-like greta array. A static diagonal matrix can always be created with e.g. `diag(3)`, and then converted into a greta array.

Usage

```
# extract
x[i]
x[i, j, ..., drop = FALSE]
head(x, n = 6L, ...)
tail(x, n = 6L, ...)
diag(x, nrow, ncol)

# replace
x[i] <- value
x[i, j, ...] <- value
diag(x) <- value

# combine
cbind(...)
rbind(...)
abind(...)
c(..., recursive = FALSE)
rep(x, times, ..., recursive = FALSE)

# get and set dimensions
length(x)
dim(x)
dim(x) <- value
```

Examples

```
## Not run:

 x <- as_data(matrix(1:12, 3, 4))

 # extract and replace
```

```
  x[1:3, ]
  x[, 2:4] <- 1:9
  e <- diag(x)
  diag(x) <- e + 1

  # combine
  cbind(x[, 2], x[, 1])
  rbind(x[1, ], x[3, ])
  abind(x[1, ], x[3, ], along = 1)
  c(x[, 1], x)
  rep(x[, 2], times = 3)

## End(Not run)
```

functions *functions for greta arrays*

Description

This is a list of functions (mostly from base R) that are currently implemented to transform greta arrays. Also see operators and transforms.

Details

TensorFlow only enables rounding to integers, so `round()` will error if `digits` is set to anything other than `0`.

Any additional arguments to `chol()`, `chol2inv`, and `solve()` will be ignored, see the TensorFlow documentation for details of these routines.

`sweep()` only works on two-dimensional greta arrays (so `MARGIN` can only be either 1 or 2), and only for subtraction, addition, division and multiplication.

`tapply()` works on column vectors (2D greta arrays with one column), and `INDEX` cannot be a greta array. Currently five functions are available, and arguments passed to ... are ignored.

Usage

```
# logarithms and exponentials
log(x)
exp(x)
log1p(x)
expm1(x)

# miscellaneous mathematics
abs(x)
mean(x)
sqrt(x)
sign(x)
```

```
# rounding of numbers
ceiling(x)
floor(x)
round(x, digits = 0)

# trigonometry
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)

# special mathematical functions
lgamma(x)
digamma(x)
choose(n, k)
lchoose(n, k)

# matrix operations
t(x)
chol(x, ...)
chol2inv(x, ...)
cov2cor(V)
solve(a, b, ...)
kronecker(X, Y, FUN = c('*', '/', '+', '-'))

# reducing operations
sum(..., na.rm = TRUE)
prod(..., na.rm = TRUE)
min(..., na.rm = TRUE)
max(..., na.rm = TRUE)

# cumulative operations
cumsum(x)
cumprod(x)

# solve an upper or lower triangular system
backsolve(r, x, k = ncol(r), upper.tri = TRUE,
          transpose = FALSE)
forwardsolve(l, x, k = ncol(l), upper.tri = FALSE,
             transpose = FALSE)

#'  # miscellaneous operations
aperm(x, perm)
apply(x, MARGIN, FUN = c("sum", "max", "mean", "min",
                         "prod", "cumsum", "cumprod"))
```

```
sweep(x, MARGIN, STATS, FUN = c('-', '+', '/', '*'))
tapply(X, INDEX, FUN = c("sum", "max", "mean", "min", "prod"), ...)
```

Examples

```
## Not run:

x <- as_data(matrix(1:9, nrow = 3, ncol = 3))
a <- log(exp(x))
b <- log1p(expm1(x))
c <- sign(x - 5)
d <- abs(x - 5)

z <- t(a)

y <- sweep(x, 1, e, '-')

## End(Not run)
```

greta — *greta: simple and scalable statistical modelling in R*

Description

greta lets you write statistical models interactively in native R code, then sample from them efficiently using Hamiltonian Monte Carlo.

The computational heavy lifting is done by TensorFlow, Google's automatic differentiation library. So greta is particularly fast where the model contains lots of linear algebra, and greta models can be run across CPU clusters or on GPUs.

See the simple example below, and take a look at the greta website for more information including tutorials and examples.

Examples

```
## Not run:
# a simple Bayesian regression model for the iris data

# priors
int <- normal(0, 5)
coef <- normal(0, 3)
sd <- lognormal(0, 3)

# likelihood
mean <- int + coef * iris$Petal.Length
distribution(iris$Sepal.Length) <- normal(mean, sd)

# build and sample
m <- model(int, coef, sd)
```

```
draws <- mcmc(m, n_samples = 100)

## End(Not run)
```

inference *statistical inference on greta models*

Description

Carry out statistical inference on greta models by MCMC or likelihood/posterior optimisation.

Usage

```
mcmc(model, sampler = hmc(), n_samples = 1000, thin = 1,
  warmup = 1000, chains = 4, n_cores = NULL, verbose = TRUE,
  pb_update = 50, one_by_one = FALSE, initial_values = initials())

stashed_samples()

extra_samples(draws, n_samples = 1000, thin = 1, n_cores = NULL,
  verbose = TRUE, pb_update = 50, one_by_one = FALSE)

initials(...)

opt(model, optimiser = bfgs(), max_iterations = 100,
  tolerance = 1e-06, initial_values = initials(), adjust = TRUE,
  hessian = FALSE)
```

Arguments

`model`	greta_model object
`sampler`	sampler used to draw values in MCMC. See `samplers` for options.
`n_samples`	number of MCMC samples to draw per chain (after any warm-up, but before thinning)
`thin`	MCMC thinning rate; every `thin` samples is retained, the rest are discarded
`warmup`	number of samples to spend warming up the mcmc sampler (moving chains toward the highest density area and tuning sampler hyperparameters).
`chains`	number of MCMC chains to run
`n_cores`	the maximum number of CPU cores used by each sampler (see details).
`verbose`	whether to print progress information to the console
`pb_update`	how regularly to update the progress bar (in iterations). If `pb_update` is less than or equal to `thin`, it will be set to `thin + 1` to ensure at least one saved iteration per `pb_update` iterations.
`one_by_one`	whether to run TensorFlow MCMC code one iteration at a time, so that greta can handle numerical errors as 'bad' proposals (see below).

`initial_values`	an optional `initials` object (or list of `initials` objects of length `chains`) giving initial values for some or all of the variables in the model. These will be used as the starting point for sampling/optimisation.
`draws`	an mcmc.list object returned by `mcmc` or `stashed_samples`
`...`	named numeric values, giving initial values of some or all of the variables in the model (unnamed variables will be automatically initialised)
`optimiser`	an `optimiser` object giving the optimisation algorithm and parameters See `optimisers`.
`max_iterations`	the maximum number of iterations before giving up
`tolerance`	the numerical tolerance for the solution, the optimiser stops when the (absolute) difference in the joint density between successive iterations drops below this level
`adjust`	whether to account for Jacobian adjustments in the joint density. Set to `FALSE` (and do not use priors) for maximum likelihood estimates, or `TRUE` for maximum *a posteriori* estimates.
`hessian`	whether to return a list of *analytically* differentiated Hessian arrays for the parameters

Details

For `mcmc()` if `verbose = TRUE`, the progress bar shows the number of iterations so far and the expected time to complete the phase of model fitting (warmup or sampling). Occasionally, a proposed set of parameters can cause numerical instability (I.e. the log density or its gradient is `NA`, `Inf` or `-Inf`); normally because the log joint density is so low that it can't be represented as a floating point number. When this happens, the progress bar will also display the proportion of proposals so far that were 'bad' (numerically unstable) and therefore rejected. Some numerical instability during the warmup phase is normal, but 'bad' samples during the sampling phase can lead to bias in your posterior sample. If you only have a few bad samples (<10\ usually resolve this with a longer warmup period or by manually defining starting values to move the sampler into a more reasonable part of the parameter space. If you have more samples than that, it may be that your model is misspecified. You can often diagnose this by using `calculate()` to evaluate the values of greta arrays, given fixed values of model parameters, and checking the results are what you expect.

greta runs multiple chains simultaneously with a single sampler, vectorising all operations across the chains. E.g. a scalar addition in your model is computed as an elementwise vector addition (with vectors having length `chains`), a vector addition is computed as a matrix addition etc. TensorFlow is able to parallelise these operations, and this approach reduced computational overheads, so this is the most efficient of computing on multiple chains.

Multiple mcmc samplers (each of which can simultaneously run multiple chains) can also be run in parallel by setting the execution plan with the `future` package. Only `plan(multisession)` futures or `plan(cluster)` futures that don't use fork clusters are allowed, since forked processes conflict with TensorFlow's parallelism. Explicitly parallelising chains on a local machine with `plan(multisession)` will probably be slower than running multiple chains simultaneously in a single sampler (with `plan(sequential)`, the default) because of the overhead required to start new sessions. However, `plan(cluster)` can be used to run chains on a cluster of machines on a local or remote network. See `future::cluster` for details, and the `future.batchtools` package to set up plans on clusters with job schedulers.

If `n_cores` = `NULL` and mcmc samplers are being run sequentially, each sampler will be allowed to use all CPU cores (possibly to compute multiple chains sequentially). If samplers are being run in parallel with the `future` package, `n_cores` will be set so that `n_cores` * `future::nbrOfWorkers` is less than the number of CPU cores.

If the sampler is aborted before finishing (and `future` parallelism isn't being used), the samples collected so far can be retrieved with `stashed_samples()`. Only samples from the sampling phase will be returned.

Samples returned by `mcmc()` and `stashed_samples()` can be added to with `extra_samples()`. This continues the chain from the last value of the previous chain and uses the same sampler and model as was used to generate the previous samples. It is not possible to change the sampler or extend the warmup period.

Because `opt()` acts on a list of greta arrays with possibly varying dimension, the `par` and `hessian` objects returned by `opt()` are named lists, rather than a vector (`par`) and a matrix (`hessian`), as returned by `optim()`. Because greta arrays may not be vectors, the Hessians may not be matrices, but could be higher-dimensional arrays. To return a Hessian matrix covering multiple model parameters, you can construct your model so that all those parameters are in a vector, then split the vector up to define the model. The parameter vector can then be passed to model. See example.

Value

`mcmc`, `stashed_samples` & `extra_samples` - an `mcmc.list` object that can be analysed using functions from the coda package. This will contain mcmc samples of the greta arrays used to create `model`.

`opt` - a list containing the following named elements:

- `par` a named list of the optimal values for the greta arrays specified in `model`
- `value` the (unadjusted) negative log joint density of the model at the parameters 'par'
- `iterations` the number of iterations taken by the optimiser
- `convergence` an integer code, 0 indicates successful completion, 1 indicates the iteration limit `max_iterations` had been reached
- `hessian` (if `hessian` = `TRUE`) a named list of hessian matrices/arrays for the parameters (w.r.t. `value`)

Examples

```
## Not run:
# define a simple Bayesian model
x <- rnorm(10)
mu <- normal(0, 5)
sigma <- lognormal(1, 0.1)
distribution(x) <- normal(mu, sigma)
m <- model(mu, sigma)

# carry out mcmc on the model
draws <- mcmc(m, n_samples = 100)

# add some more samples
draws <- extra_samples(draws, 200)
```

```
#' # initial values can be passed for some or all model variables
draws <- mcmc(m, chains = 1, initial_values = initials(mu = -1))

# if there are multiple chains, a list of initial values should be passed,
# othewise the same initial values will be used for all chains
inits <- list(initials(sigma = 0.5), initials(sigma = 1))
draws <- mcmc(m, chains = 2, initial_values = inits)

# you can auto-generate a list of initials with something like this:
inits <- replicate(4,
                   initials(mu = rnorm(1), sigma = runif(1)),
                   simplify = FALSE)
draws <- mcmc(m, chains = 4, initial_values = inits)

# or find the MAP estimate
opt_res <- opt(m)

# get the MLE of the normal variance
mu <- variable()
variance <- variable(lower = 0)
distribution(x) <- normal(mu, sqrt(variance))
m2 <- model(variance)

# adjust = FALSE skips the jacobian adjustments used in MAP estimation, to
# give the true maximum likelihood estimates
o <- opt(m2, adjust = FALSE)

# the MLE corresponds to the *unadjusted* sample variance, but differs
# from the sample variance
o$par
mean((x - mean(x)) ^ 2)  # same
var(x)  # different

# initial values can also be passed to optimisers:
o <- opt(m2, initial_values = initials(variance = 1))

# and you can return a list of the Hessians for each of these parameters
o <- opt(m2, hessians = TRUE)
o$hessians


# to get a hessian matrix across multiple greta arrays, you must first
# combine them and then split them up for use in the model (so that the
# combined vector is part of the model) and pass that vector to model:
params <- c(variable(), variable(lower = 0))
mu <- params[1]
variance <- params[2]
distribution(x) <- normal(mu, sqrt(variance))
m3 <- model(params)
o <- opt(m3, hessians = TRUE)
o$hessians
```

```
## End(Not run)
```

`internals` *internal greta methods*

Description

A list of functions and R6 class objects that can be used to develop extensions to greta. Most users will not need to access these methods, and it is not recommended to use them directly in model code.

Details

This help file lists the available internals, but they are not fully documented and are subject to change and deprecation without warning (though care will be taken not to break dependent packages on CRAN). For an overview of how greta works internally, see the *technical details* vignette. See `https://github.com/greta-dev` for examples of R packages extending and building on greta.

Please get in contact via GitHub if you want to develop an extension to greta and need more details of how to use these internal functions.

You can use `attach()` to put a sublist in the search path. E.g. `attach(.internals$nodes$constructors)` will enable you to call `op()`, `vble()` and `distrib()` directly.

Usage

```
.internals$greta_arrays$unknowns        # greta array print methods
.internals$inference$progress_bar       # progress bar tools
                     samplers           # MCMC samplers
                     stash              # stashing MCMC samples
.internals$nodes$constructors           # node creation wrappers
                 distribution_classes   # R6 distribution classes
                 mixture_classes        # R6 mixture distribution classes
                 node_classes           # R6 node classes
.internals$tensors                      # functions on tensors
.internals$utils$checks                 # checking function inputs
                 colours                # greta colour scheme
                 dummy_arrays           # mocking up extract/replace
                 misc                   # code simplification etc.
                 samplers               # mcmc helpers
.internals$greta_stash                  # internal information storage
```

joint — *define joint distributions*

Description

`joint` combines univariate probability distributions together into a multivariate (and *a priori* independent between dimensions) joint distribution, either over a variable, or for fixed data.

Usage

```
joint(..., dim = NULL)
```

Arguments

`...`	variable greta arrays following probability distributions (see `distributions`); the components of the joint distribution.
`dim`	the dimensions of the greta array to be returned, either a scalar or a vector of positive integers. The final dimension of the greta array returned will be determined by the number of component distributions

Details

The component probability distributions must all be either continuous or discrete, and must have the same dimensions.

This functionality is unlikely to be useful in most models, since the same result can usually be achieved by combining variables with separate distributions. It is included for situations where it is more convenient to consider these as a single distribution, e.g. for use with `distribution` or `mixture`.

Examples

```
## Not run:
# an uncorrelated bivariate normal
x <- joint(normal(-3, 0.5), normal(3, 0.5))
m <- model(x)
plot(mcmc(m, n_samples = 500))

# joint distributions can be used to define densities over data
x <- cbind(rnorm(10, 2, 0.5), rbeta(10, 3, 3))
mu <- normal(0, 10)
sd <- normal(0, 3, truncation = c(0, Inf))
a <- normal(0, 3, truncation = c(0, Inf))
b <- normal(0, 3, truncation = c(0, Inf))
distribution(x) <- joint(normal(mu, sd), beta(a, b),
                         dim = 10)
m <- model(mu, sd, a, b)
plot(mcmc(m))

## End(Not run)
```

`mixture` *mixtures of probability distributions*

Description

`mixture` combines other probability distributions into a single mixture distribution, either over a variable, or for fixed data.

Usage

```
mixture(..., weights, dim = NULL)
```

Arguments

`...`	variable greta arrays following probability distributions (see `distributions`); the component distributions in a mixture distribution.
`weights`	a column vector or array of mixture weights, which must be positive, but need not sum to one. The first dimension must be the number of distributions, the remaining dimensions must either be 1 or match the distribution dimension.
`dim`	the dimensions of the greta array to be returned, either a scalar or a vector of positive integers.

Details

The `weights` are rescaled to sum to one along the first dimension, and are then used as the mixing weights of the distribution. *Ie.* the probability density is calculated as a weighted sum of the component probability distributions passed in via `...`

The component probability distributions must all be either continuous or discrete, and must have the same dimensions.

Examples

```
## Not run:
# a scalar variable following a strange bimodal distibution
weights <- uniform(0, 1, dim = 3)
a <- mixture(normal(-3, 0.5),
             normal(3, 0.5),
             normal(0, 3),
             weights = weights)
m <- model(a)
plot(mcmc(m, n_samples = 500))

# simulate a mixture of poisson random variables and try to recover the
# parameters with a Bayesian model
x <- c(rpois(800, 3),
       rpois(200, 10))

weights <- uniform(0, 1, dim = 2)
```

```
rates <- normal(0, 10, truncation = c(0, Inf), dim = 2)
distribution(x) <- mixture(poisson(rates[1]),
                           poisson(rates[2]),
                           weights = weights)
m <- model(rates)
draws_rates <- mcmc(m, n_samples = 500)

# check the mixing probabilities after fitting using calculate()
# (you could also do this within the model)
normalized_weights <- weights / sum(weights)
draws_weights <- calculate(normalized_weights, draws_rates)

# get the posterior means
summary(draws_rates)$statistics[, "Mean"]
summary(draws_weights)$statistics[, "Mean"]

# weights can also be an array, giving different mixing weights
# for each observation (first dimension must be number of components)
dim <- c(5, 4)
weights <- uniform(0, 1, dim = c(2, dim))
b <- mixture(normal(1, 1, dim = dim),
             normal(-1, 1, dim = dim),
             weights = weights)

## End(Not run)
```

`model` *greta model objects*

Description

Create a `greta_model` object representing a statistical model (using `model`), and plot a graphical representation of the model. Statistical inference can be performed on `greta_model` objects with `mcmc`

Usage

```
model(..., precision = c("double", "single"), compile = TRUE)

## S3 method for class 'greta_model'
print(x, ...)

## S3 method for class 'greta_model'
plot(x, y, colour = "#996bc7", ...)
```

Arguments

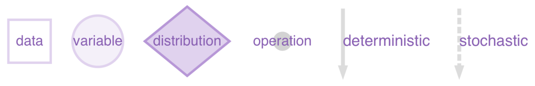
`...`	for `model`: `greta_array` objects to be tracked by the model (i.e. those for which samples will be retained during mcmc). If not provided, all of the non-data

	`greta_array` objects defined in the calling environment will be tracked. For `print` and `plot`:further arguments passed to or from other methods (currently ignored).
`precision`	the floating point precision to use when evaluating this model. Switching from `"double"` (the default) to `"single"` may decrease the computation time but increase the risk of numerical instability during sampling.
`compile`	whether to apply XLA JIT compilation to the TensorFlow graph representing the model. This may slow down model definition, and speed up model evaluation.
`x`	a `greta_model` object
`y`	unused default argument
`colour`	base colour used for plotting. Defaults to `greta` colours in violet.

Details

`model()` takes greta arrays as arguments, and defines a statistical model by finding all of the other greta arrays on which they depend, or which depend on them. Further arguments to `model` can be used to configure the TensorFlow graph representing the model, to tweak performance.

The plot method produces a visual representation of the defined model. It uses the `DiagrammeR` package, which must be installed first. Here's a key to the plots:



Value

`model` - a `greta_model` object.

`plot` - a `DiagrammeR::grViz` object, with the `DiagrammeR::dgr_graph` object used to create it as an attribute `"dgr_graph"`.

Examples

```
## Not run:

# define a simple model
mu <- variable()
sigma <- normal(0, 3, truncation = c(0, Inf))
x <- rnorm(10)
distribution(x) <- normal(mu, sigma)

m <- model(mu, sigma)

plot(m)

## End(Not run)
```

operators *arithmetic, logical and relational operators for greta arrays*

Description

This is a list of currently implemented arithmetic, logical and relational operators to combine greta arrays into probabilistic models. Also see functions and transforms.

Details

greta's operators are used just like R's the standard arithmetic, logical and relational operators, but they return other greta arrays. Since the operations are only carried during sampling, the greta array objects have unknown values.

Usage

```
# arithmetic operators
-x
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
x %*% y

# logical operators
!x
x & y
x | y

# relational operators
x < y
x > y
x <= y
x >= y
x == y
x != y
```

Examples

```
## Not run:

x <- as_data(-1:12)

# arithmetic
```

```
a <- x + 1
b <- 2 * x + 3
c <- x %% 2
d <- x %/% 5

# logical
e <- (x > 1) | (x < 1)
f <- e & (x < 2)
g <- !f

# relational
h <- x < 1
i <- (-x) >= x
j <- h == x

## End(Not run)
```

`optimisers` *optimisation methods*

Description

Functions to set up optimisers (which find parameters that maximise the joint density of a model) and change their tuning parameters, for use in `opt()`. For details of the algorithms and how to tune them, see the SciPy optimiser docs or the TensorFlow optimiser docs.

Usage

```
nelder_mead()

powell()

cg()

bfgs()

newton_cg()

l_bfgs_b(maxcor = 10, maxls = 20)

tnc(max_cg_it = -1, stepmx = 0, rescale = -1)

cobyla(rhobeg = 1)

slsqp()

gradient_descent(learning_rate = 0.01)
```

```
adadelta(learning_rate = 0.001, rho = 1, epsilon = 1e-08)

adagrad(learning_rate = 0.8, initial_accumulator_value = 0.1)

adagrad_da(learning_rate = 0.8, global_step = 1L,
  initial_gradient_squared_accumulator_value = 0.1,
  l1_regularization_strength = 0, l2_regularization_strength = 0)

momentum(learning_rate = 0.001, momentum = 0.9, use_nesterov = TRUE)

adam(learning_rate = 0.1, beta1 = 0.9, beta2 = 0.999,
  epsilon = 1e-08)

ftrl(learning_rate = 1, learning_rate_power = -0.5,
  initial_accumulator_value = 0.1, l1_regularization_strength = 0,
  l2_regularization_strength = 0)

proximal_gradient_descent(learning_rate = 0.01,
  l1_regularization_strength = 0, l2_regularization_strength = 0)

proximal_adagrad(learning_rate = 1, initial_accumulator_value = 0.1,
  l1_regularization_strength = 0, l2_regularization_strength = 0)

rms_prop(learning_rate = 0.1, decay = 0.9, momentum = 0,
  epsilon = 1e-10)
```

Arguments

`maxcor`	maximum number of 'variable metric corrections' used to define the approximation to the hessian matrix
`maxls`	maximum number of line search steps per iteration
`max_cg_it`	maximum number of hessian * vector evaluations per iteration
`stepmx`	maximum step for the line search
`rescale`	log10 scaling factor used to trigger rescaling of objective
`rhobeg`	reasonable initial changes to the variables
`learning_rate`	the size of steps (in parameter space) towards the optimal value
`rho`	the decay rate
`epsilon`	a small constant used to condition gradient updates
`initial_accumulator_value`	initial value of the 'accumulator' used to tune the algorithm
`global_step`	the current training step number
`initial_gradient_squared_accumulator_value`	initial value of the accumulators used to tune the algorithm
`l1_regularization_strength`	L1 regularisation coefficient (must be 0 or greater)

`l2_regularization_strength`	L2 regularisation coefficient (must be 0 or greater)
`momentum`	the momentum of the algorithm
`use_nesterov`	whether to use Nesterov momentum
`beta1`	exponential decay rate for the 1st moment estimates
`beta2`	exponential decay rate for the 2nd moment estimates
`learning_rate_power`	power on the learning rate, must be 0 or less
`decay`	discounting factor for the gradient

Details

The `cobyla()` does not provide information about the number of iterations nor convergence, so these elements of the output are set to NA

Value

an `optimiser` object that can be passed to `opt`.

Examples

```
## Not run:
# use optimisation to find the mean and sd of some data
x <- rnorm(100, -2, 1.2)
mu <- variable()
sd <- variable(lower = 0)
distribution(x) <- normal(mu, sd)
m <- model(mu, sd)

# configure optimisers & parameters via 'optimiser' argument to opt
opt_res <- opt(m, optimiser = bfgs())

# compare results with the analytic solution
opt_res$par
c(mean(x), sd(x))

## End(Not run)
```

`overloaded` *Functions overloaded by greta*

Description

greta provides a wide range of methods to apply common R functions and operations to `greta_array` objects. A few of these functions and operators are not associated with a class system, so they are overloaded here. This should not affect normal use of these functions, but they need to be documented to satisfy CRAN's check.

Usage

```
x %*% y

chol2inv(x, size = NCOL(x), LINPACK = FALSE)

cov2cor(V)

identity(x)

colMeans(x, na.rm = FALSE, dims = 1L)

rowMeans(x, na.rm = FALSE, dims = 1L)

colSums(x, na.rm = FALSE, dims = 1L)

rowSums(x, na.rm = FALSE, dims = 1L)

sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)

backsolve(r, x, k = ncol(r), upper.tri = TRUE, transpose = FALSE)

forwardsolve(l, x, k = ncol(l), upper.tri = FALSE, transpose = FALSE)

apply(X, MARGIN, FUN, ...)

tapply(X, INDEX, FUN, ...)

eigen(x, symmetric, only.values, EISPACK)

rdist(x1, x2 = NULL, compact = FALSE)

abind(..., along = N, rev.along = NULL, new.names = NULL,
  force.array = TRUE, make.names = use.anon.names,
  use.anon.names = FALSE, use.first.dimnames = FALSE,
  hier.names = FALSE, use.dnns = FALSE)

diag(x = 1, nrow, ncol)
```

Arguments

`x, y, size, LINPACK, V, na.rm, dims, MARGIN, STATS, FUN, check.margin, ..., r, k, upper.tri, transpose, l, X`
arguments as in original documentation

samplers *MCMC samplers*

Description

Functions to set up MCMC samplers and change the starting values of their parameters, for use in `mcmc()`.

Usage

```
hmc(Lmin = 5, Lmax = 10, epsilon = 0.1, diag_sd = 1)

rwmh(proposal = c("normal", "uniform"), epsilon = 0.1, diag_sd = 1)

slice(max_doublings = 5)
```

Arguments

`Lmin`	minimum number of leapfrog steps (positive integer, Lmin > Lmax)
`Lmax`	maximum number of leapfrog steps (positive integer, Lmax > Lmin)
`epsilon`	leapfrog stepsize hyperparameter (positive, will be tuned)
`diag_sd`	estimate of the posterior marginal standard deviations (positive, will be tuned).
`proposal`	the probability distribution used to generate proposal states
`max_doublings`	the maximum number of iterations of the 'doubling' algorithm used to adapt the size of the slice

Details

During the warmup iterations of `mcmc`, some of these sampler parameters will be tuned to improve the efficiency of the sampler, so the values provided here are used as starting values.

For `hmc()`, the number of leapfrog steps at each iteration is selected uniformly at random from between `Lmin` and `Lmax`. `diag_sd` is used to rescale the parameter space to make it more uniform, and make sampling more efficient.

`rwmh()` creates a random walk Metropolis-Hastings sampler; a a gradient-free sampling algorithm. The algorithm involves a proposal generating step 'proposal_state = current_state + perturb' by a random perturbation, followed by Metropolis-Hastings accept/reject step. The class is implemented for uniform and normal proposals.

`slice()` implements a multivariate slice sampling algorithm. Currently this algorithm can only be used with single-precision models (set using the `precision` argument to `model`). The parameter `max_doublings` is not tuned during warmup.

Value

a `sampler` object that can be passed to `mcmc`.

structures *create data greta arrays*

Description

These structures can be used to set up more complex models. For example, scalar parameters can be embedded in a greta array by first creating a greta array with `zeros()` or `ones()`, and then embedding the parameter value using greta's replacement syntax.

Usage

```
zeros(...)

ones(...)

greta_array(data = 0, dim = length(data))
```

Arguments

`...`	dimensions of the greta arrays to create
`data`	a vector giving data to fill the greta array. Other object types are coerced by `as.vector`.
`dim`	an integer vector giving the dimensions for the greta array to be created.

Details

`greta_array` is a convenience function to create an R array with `array` and then coerce it to a greta array. I.e. when passed something that can be coerced to a numeric array, it is equivalent to `as_data(array(data,dim))`.

If `data` is a greta array and dim is different than `dim(data)`, a reshaped greta array is returned. This is equivalent to: `dim(data) <-dim`.

Value

a greta array object

Examples

```
## Not run:

# a 3 row, 4 column greta array of 0s
z <- zeros(3, 4)

# a 3x3x3 greta array of 1s
z <- ones(3, 3, 3)

# a 2x4 greta array filled with pi
```

```
z <- greta_array(pi, dim = c(2, 4))

# a 3x3x3 greta array filled with 1, 2, and 3
z <- greta_array(1:3, dim = c(3, 3, 3))

## End(Not run)
```

`transforms` *transformation functions for greta arrays*

Description

transformations for greta arrays, which may also be used as inverse link functions. Also see operators and functions.

Usage

```
iprobit(x)

ilogit(x)

icloglog(x)

icauchit(x)

log1pe(x)

imultilogit(x)
```

Arguments

`x`	a real-valued (i.e. values ranging from -Inf to Inf) greta array to transform to a constrained value

Details

greta does not allow you to state the transformation/link on the left hand side of an assignment, as is common in the BUGS and STAN modelling languages. That's because the same syntax has a very different meaning in R, and can only be applied to objects that are already in existence. The inverse forms of the common link functions (prefixed with an 'i') can be used instead.

The `log1pe` inverse link function is equivalent to `log(1 + exp(x))`, yielding a positive transformed parameter. Unlike the log transformation, this transformation is approximately linear for x > 1. i.e. when $x > 1$, $y \approx x$

`imultilogit` expects an n-by-m greta array, and returns an n-by-(m+1) greta array of positive reals whose rows sum to one. This is equivalent adding a final column of 0s and then running the softmax function widely used in machine learning.

Examples

```
## Not run:

 x1 <- normal(1, 3, dim = 10)

 # transformation to the unit interval
 p1 <- iprobit(x1)
 p2 <- ilogit(x1)
 p3 <- icloglog(x1)
 p4 <- icauchit(x1)

 # and to positive reals
 y <- log1pe(x1)

 # transform from 10x3 to 10x4, where rows are a complete set of
 # probabilities
 x2 <- normal(1, 3, dim = c(10, 3))
 z <- imultilogit(x2)


## End(Not run)
```

---

variable *create greta variables*

Description

`variable()` creates greta arrays representing unknown parameters, to be learned during model fitting. These parameters are not associated with a probability distribution. To create a variable greta array following a specific probability distribution, see `distributions`.

Usage

```
variable(lower = -Inf, upper = Inf, dim = 1)
```

Arguments

`lower, upper`	scalar values giving optional limits to variables. These must be specified as numerics, they cannot be greta arrays (though see details for a workaround). They can be set to `-Inf` (`lower`) or `Inf` (`upper`), though `lower` must always be less than `upper`.
`dim`	the dimensions of the greta array to be returned, either a scalar or a vector of positive integers. See details.

Details

`lower` and `upper` must be fixed, they cannot be greta arrays. This ensures these values can always be transformed to a continuous scale to run the samplers efficiently. However, a variable parameter with dynamic limits can always be created by first defining a variable constrained between 0 and 1, and then transforming it to the required scale. See below for an example.

Examples

```
## Not run:

# a scalar variable
a <- variable()

# a positive length-three variable
b <- variable(lower = 0, dim = 3)

# a 2x2x2 variable bounded between 0 and 1
c <- variable(lower = 0, upper = 1, dim = c(2, 2, 2))

# create a variable, with lower and upper defined by greta arrays
min <- as_data(iris$Sepal.Length)
max <- min ^ 2
d <- min + variable(0, 1, dim = nrow(iris)) * (max - min)

## End(Not run)
```

Index