Package 'BayesMVP'

February 25, 2025

Type Package

Title BayesMVP: A high-performance implementation of adaptive diffusion-space HMC for multivariate probit models and other models with high-dimensional latent variables.

Version 1.0

Date 2024-05-18

Author Enzo Cerullo

Maintainer Enzo Cerullo <enzo.cerullo@bath.edu>

Description BayesMVP uses diffusion-space Hamiltonian Monte Carlo (HMC) to efficiently sample the multivariate probit model (MVP) to model correlated binary data, as well as the latent class MVP model for the evaluation of test accuracy data without a gold standard. It achieves rapid sampling through manually-derived gradients and chunking. Users can also input any Stan model for efficient sampling of high dimensional latent variable vectors.

License GPL-3

Imports Rcpp (>= 1.0.10), RcppEigen, RcppParallel, dqrng, cmdstanr, bridgestan, bayesplot, posterior, dplyr, ggplot2, tibble, tictoc

LinkingTo Rcpp, RcppEigen, RcppParallel, dqrng

Suggests roxygen2

Roxygen list(markdown = TRUE, roclets = c(``namespace'', ``rd''))

SystemRequirements GNU make, C++17

RoxygenNote 7.3.2

Encoding UTF-8

Depends R (>= 3.5.0)

Contents

`bridgestan_path` *bridgestan_path*

Description

bridgestan_path

Usage

```
bridgestan_path()
```

`cmdstanr_path` *cmdstanr_path*

Description

cmdstanr_path

Usage

```
cmdstanr_path()
```

`MVP_class_extract_and_plot`
MVP extract and plot Class

Description

Class for handling MVP model summaries, samples (trace) and diagnostics.

Details

This class is called via the main "MVP_model" R6 class.

Model Types

- MVP: Multivariate probit model
- LC-MVP: Latent class multivariate probit model
- latent_trait: Latent trait model
- Stan: User-supplied Stan models (files extension must be .stan).

Class Relationship

This class is not meant to be instantiated/called directly by users. Instead, it is created and returned by the MVP_model$summary() method. It provides methods for:

- Extracting MCMC diagnostics (divergences, ESS, etc.)
- Creating trace and density plots
- Computing efficiency metrics
- Accessing posterior draws in various formats

Common Usage Patterns

Basic Diagnostics get_divergences() -> get_efficiency_metrics()

Parameter Summaries get_summary_main() -> get_summary_transformed()

Visualization plot_traces() -> plot_densities()

Custom Analysis get_posterior_draws() -> Your analysis

Public fields

`summary_object` The summary object, obtained from the main "MVP_model" R6 class.

`init_object` Initialization object, obtained from the main "MVP_model" R6 class.

`n_nuisance` The total number of nuisance parameters, please see the "MVP_model" R6 class documentation for more details.

Methods

Public methods:

- `MVP_class_extract_and_plot$new()`
- `MVP_class_extract_and_plot$get_divergences()`
- `MVP_class_extract_and_plot$get_summary_main()`
- `MVP_class_extract_and_plot$get_summary_transformed()`
- `MVP_class_extract_and_plot$get_summary_generated_quantities()`
- `MVP_class_extract_and_plot$get_posterior_draws()`
- `MVP_class_extract_and_plot$get_all_traces()`
- `MVP_class_extract_and_plot$get_log_lik_trace()`
- `MVP_class_extract_and_plot$get_nuisance_trace()`
- `MVP_class_extract_and_plot$get_posterior_draws_as_tibbles()`
- `MVP_class_extract_and_plot$get_efficiency_metrics()`
- `MVP_class_extract_and_plot$get_HMC_info()`
- `MVP_class_extract_and_plot$time_to_target_ESS()`
- `MVP_class_extract_and_plot$iter_to_target_ESS()`
- `MVP_class_extract_and_plot$plot_traces()`
- `MVP_class_extract_and_plot$plot_densities()`
- `MVP_class_extract_and_plot$clone()`

Method `new()`**:**

Usage:

```
MVP_class_extract_and_plot$new(model_summary, init_object, n_nuisance)
```

Arguments:

`model_summary` The summary object, obtained from the main "MVP_model" R6 class.

`init_object` The Initialization object, obtained from the main "MVP_model" R6 class.

`n_nuisance` The total number of nuisance parameters, please see the "MVP_model" R6 class documentation for more details.

Method `get_divergences()`**:**

Usage:

```
MVP_class_extract_and_plot$get_divergences()
```

Returns: Returns self$summary_object$summaries$divergences obtained from the main "MVP_model" R6 class. self$summary_object$summaries$divergences contains divergence information (e.g. % of transitions which diverged).

Method `get_summary_main()`**:**

Usage:

```
MVP_class_extract_and_plot$get_summary_main()
```

Returns: Returns main parameter summaries as a tibble.

Method `get_summary_transformed()`**:**

Usage:

```
MVP_class_extract_and_plot$get_summary_transformed()
```

Returns: Returns transformed parameter summaries as a tibble.

Method `get_summary_generated_quantities()`**:**

Usage:

```
MVP_class_extract_and_plot$get_summary_generated_quantities()
```

Returns: Returns generated quantities as a tibble. Convenience method for posterior draws (as 3D arrays)

Method `get_posterior_draws()`**:**

Usage:

```
MVP_class_extract_and_plot$get_posterior_draws()
```

Returns: Returns posterior draws as a 3D array (iterations, chains, parameters). Convenience method for all traces (as 3D arrays)

Method `get_all_traces()`**:**

Usage:

```
MVP_class_extract_and_plot$get_all_traces()
```

Returns: Returns posterior traces as a 3D arrays (iterations, chains, parameters). Convenience method for posterior draws for log_lik (as 3D arrays)

Method `get_log_lik_trace()`**:**

Usage:

```
MVP_class_extract_and_plot$get_log_lik_trace()
```

Returns: Returns posterior draws for log_lik as a 3D array (iterations, chains, parameters). Convenience method for posterior draws for nuisance (as 3D arrays)

Method `get_nuisance_trace()`**:**

Usage:

```
MVP_class_extract_and_plot$get_nuisance_trace()
```

Returns: Returns posterior draws for nuisance as a 3D array (iterations, chains, parameters). Convenience method for posterior draws (as tibbles)

Method `get_posterior_draws_as_tibbles()`**:**

Usage:

```
MVP_class_extract_and_plot$get_posterior_draws_as_tibbles()
```

Returns: Returns trace as tibbles. The trace gets returned as 3 seperate tibbles (one tibble for the main parameters, one for the transformed parameters, and one for generated quantities).

Method `get_efficiency_metrics()`**:**

Usage:

```
MVP_class_extract_and_plot$get_efficiency_metrics()
```

Returns: Returns a list containing efficiency metrics including:

- time_burnin: Time spent in burnin phase
- time_sampling: Time spent sampling
- time_total_MCMC: Total time spent doing MCMC/HMC sampling, excluding time to compute summaries and diagnostics.
- time_total_inc_summaries: Total time spent doing MCMC/HMC sampling, including time to compute summaries and diagnostics.
- Min_ESS_main: The minimum ESS of the main model parameters.
- Min_ESS_per_sec_sampling: The minimum ESS per second for the sampling phase.
- Min_ESS_per_sec_total: The minimum ESS per second for the total model run time, including any time spent computing summaries and diagnostics.
- Min_ESS_per_grad_sampling: The minimum ESS per gradient evaluation for the sampling phase.
- grad_evals_per_sec: The number of gradient evaluations performed per second.
- est_time_to_100_ESS_sampling: The estimated sampling time to reach a minimum ESS of 100.
- est_time_to_1000_ESS_sampling: The estimated sampling time to reach a minimum ESS of 1000.
- est_time_to_10000_ESS_sampling: The estimated sampling time to reach a minimum ESS of 10,000.
- est_time_to_100_ESS_wo_summaries: The estimated total time (expluding time to compute model summaries and diagnostics) to reach a minimum ESS of 100.

- est_time_to_1000_ESS_wo_summaries: The estimated total time (expluding time to compute model summaries and diagnostics) to reach a minimum ESS of 1000.
- est_time_to_10000_ESS_wo_summaries: The estimated total time (expluding time to compute model summaries and diagnostics) to reach a minimum ESS of 10,000.
- est_time_to_100_ESS_inc_summaries: The estimated total time (including time spent computing model summaries and diagnostics) to reach a minimum ESS of 100.
- est_time_to_1000_ESS_inc_summaries: The estimated total time (including time spent computing model summaries and diagnostics) to reach a minimum ESS of 1000.
- est_time_to_10000_ESS_inc_summaries: The estimated total time (including time spent computing model summaries and diagnostics) to reach a minimum ESS of 10,000.

Method `get_HMC_info()`:

Usage:

```
MVP_class_extract_and_plot$get_HMC_info()
```

Returns: Returns a list containing HMC algorithm metrics including:

- tau_main: The HMC path length (τ) for sampling the main parameters.
- eps_main: The HMC step-size (ϵ) for sampling the main parameters.
- tau_us: The HMC path length (τ) for sampling the nuisance parameters.
- eps_us: The HMC step-size (ϵ) for sampling the nuisance parameters.
- n_chains_sampling: The number of parallel chains used during the sampling phase.
- n_chains_burnin: The number of parallel chains used during the burnin phase.
- n_iter: The number of iterations used during the sampling phase.
- n_burnin: The number of iterations used during the burnin phase.
- LR_main: The ADAM learning rate (main parameters).
- LR_us: The ADAM learning rate (nuisance parameters).
- adapt_delta: The target Metropolis-Hastings acceptance probability.
- metric_type_main: The type of HMC metric used for main parameters (Hessian or Empirical).
- metric_shape_main: The shape of HMC metric used for main parameters (dense or diag).
- metric_type_nuisance: The type of HMC metric used for the nuisance parameters (Hessian or Empirical).
- metric_shape_nuisance: The shape of HMC metric used for the nuisance parameters (dense or diag).
- diffusion_HMC: Whether diffusion HMC was used to sample the nuisance parameters or not.
- partitioned_HMC: Whether partitioned HMC was used or not (if `FALSE`, then parameters were sampled all at once, and if `TRUE`, then the nuisance are sampled conditional on the main parameters).
- n_superchains: The number of superchains.
- interval_width_main: The interval width for the main parameters. The metric is computed at the end of each interval.
- interval_width_nuisance: The interval width for the nuisance parameters. The metric is computed at the end of each interval.
- force_autodiff: Whether autodiff was used (forced) - only relevant for built-in models, as Stan models always use autodiff.

- force_PartialLog: Whether a partial-log-scale version of the model was used (forced) - only relevant for built-in models.
- multi_attempts: Whether multiple attempts were made when evaluating the lp_grad function (i.e. first try normal param., then partial-log-scale, and finally autodiff + partial-log-scale) - only relevant for built-in models.

Method `time_to_target_ESS()`:

Usage:

```
MVP_class_extract_and_plot$time_to_target_ESS(target_ESS)
```

Arguments:

`target_ESS` The target ESS.

Returns: Returns a list called "target_ESS_times" which contains the estimated sampling time to reach the target ESS ("sampling_time_to_target_ESS"), the estimated total time excluding the time it takes to compute model summaries ("total_time_to_target_ESS_wo_summaries"), and the total estimated time to reach the target ESS ("total_time_to_target_ESS_with_summaries").

Method `iter_to_target_ESS()`:

Usage:

```
MVP_class_extract_and_plot$iter_to_target_ESS(target_ESS)
```

Arguments:

`target_ESS` The target ESS.

Returns: Returns a list called "target_ESS_iter" which contains the estimated number of sampling iterations (n_iter) to reach the target ESS ("sampling_iter_to_target_ESS").

Method `plot_traces()`:

Usage:

```
MVP_class_extract_and_plot$plot_traces(params = NULL, batch_size = 9)
```

Arguments:

`params` The parameters to generate trace plots for. This is a character vector - e.g. to plot trace plots for beta: params = c("beta"). The default is NULL and the trace plots for all model parameters (which have a trace array) will be plotted.

`batch_size` The number of trace plots to display per panel. Default is 9.

Returns: If no parameters specified (i.e. params = NULL), then this will return an object containing the trace plots for all model parameters which have a trace array.

Method `plot_densities()`:

Usage:

```
MVP_class_extract_and_plot$plot_densities(params = NULL, batch_size = 9)
```

Arguments:

`params` The parameters to generate posterior density plots for. This is a character vector - e.g. to plot the posterior density plots for beta: params = c("beta"). The default is NULL and the posterior density plots for all model parameters (which have a trace array) will be plotted.

`batch_size` The number of posterior density plots to display per panel. Default is 9.

Returns: If no parameters specified (i.e. params = NULL), then this will return an object containing the posterior density plots for all model parameters which have a trace array.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`MVP_model` *MVP Model Class*

Description

R6 Class for MVP model initialization and sampling.

Details

MVP Model Class

This class handles:

- Model initialization and compilation
- MCMC sampling with adaptive diffusion-pathspace HMC.
- Parameter updates and diagnostics
- Support for both built-in models (MVP, LC-MVP, latent_trait) and user-supplied Stan models

Model Types

- MVP: Multivariate probit model
- LC-MVP: Latent class multivariate probit model
- latent_trait: Latent trait model
- Stan: User-supplied Stan models (files extension must be .stan).

Common Workflows

Basic Usage Initialize -> Sample -> Summary -> Plot

Parameter Updates Initialize -> Update parameters via sample() -> Sample -> Summary -> Plot

Stan Models Initialize with Stan model -> Sample -> Summary -> Plot

Public fields

`Model_type` Type of model to fit (can be one of: "MVP", "LC_MVP", "latent_trait" for built-in models, or "Stan" for fitting any Stan model via BayesMVP).

`init_object` Initialization object.

`result` Model results object.

`model_fit_object` Model fit object.

`y` The dataset. Note that for built-in models (i.e., MVP, LC_MVP, or latent_trait) y should be an N x n_outcomes matrix.

`N` The total sample size. Note that for built-in models (i.e., MVP, LC_MVP, or latent_trait) this should be equal to: $N = nrow(y)$.

`n_params_main` The total number of main model parameters (i.e., excluding nuisance parameters / high-dimensional latent variables). For Stan models (i.e., if "Model_type" is set to "Stan" and you are running a Stan model), this should equal the number of parameters that you define in the "parameters" block of the Stan model, EXCEPT for any nuisance parameters (i.e., high-dimensional latent variables).

`n_nuisance` The total number of nuisance parameters, i.e. the dimension of the high-dimensional latent variable vector. For built-in models (i.e., MVP, LC_MVP, or latent_trait), this should be equal to: $n_nuisance = N \cdot n_outcomes = nrow(y) \cdot ncol(y)$.

`init_lists_per_chain` A list of dimension n_chains_burnin (NOT n_chains_sampling, unless n_chains_sampling = n_chains_burnin), where each element of the list is another list which contains the # initial values for the chain. Note that this is the same format for initial values that Stan (both rstan and cmdstanr) uses.

`model_args_list` List containing model-specific arguments. This is only relevant for built-in models (i.e., MVP, LC_MVP, or latent_trait). All arguments in this list are optional.

Arguments which are relevant to all three of the built-in models include:

- `n_covariates_per_outcome`: A matrix of dimension `n_class` x `n_outcomes`, which contains the number of covariates per outcome. If your model has no covariates (which is often the case for the LC_MVP or latent_trait), this does not need to be included, and it just be a matrix of 1's (since each outcome has 1 intercept). Also note that n_class = 1 for the MVP, and for the LC_MVP and latent_trait n_class = 2.
- `prior_coeffs_mean`: An array of dimension `n_class` x `n_outcomes` x `n_covariates_max`, where `n_covariates_max` is equal to the number of covariates which are contained in the outcome which has the most covariates. This matrix contains the prior mean of each coefficient parameter. In other words, if: $\beta_{c,t,k} \sim \mathrm{N}\left(\mu_{c,t,k}, \sigma_{c,t,k}\right)$, then element (c, t, k) in the matrix corresponds to $\mu_{c,t,k}$, where c is an index for class, t is an index for outcome, and k is an index for the coefficient for outcome t. The default is an array of zeros.
- `prior_coeffs_sd`: This is the same as `prior_coeffs_mean`, except that each element in thr array corresponds to the prior SD. In other words, each element in the array corresponds to: $\sigma_{c,t,k}$, where: $\beta_{c,t,k} \sim \mathrm{N}\left(\mu_{c,t,k}, \sigma_{c,t,k}\right)$. The default is an array of ones.
- `vect_type`: The SIMD (single-instruction, multiple-data) vectorisation type to use for math functions (such as log, exp, Phi, etc). The default is AVX-512 if available, then AVX2 if not, and if neither AVX-512 nor AVX2 are available (e.g., on ARM-based CPU's such as Apple systems), then BayesMVP will use the Stan C++ math library functions, and they will decide what vectorisation type to use. More vectorisation types will be supported in the future.

- `num_chunks`: The number of chunks to use in the log-probability and gradient function. By default, the number selected will depend on your CPU.
- `Phi_type`: Type of $\Phi()$ function implementation to use, where $\Phi()$ is the standard normal CDF. Can be either "Phi" (the default) or "Phi_approx". The default is "Phi". Note that $\Phi()$ will use a fast, highly accurate polynomial approximation of $\Phi()$ if `vect_type` is either AVX-512 or AVX2. Otherwise, it will use the Phi function from the Stan math C++ library.

Arguments which are only relevant to the MVP and LC_MVP include:

- `corr_force_positive`: This will force all elements in the correlation matrix (or matrices if LC_MVP) to be positive. Uses the method proposed by Pinkney et al, 2024.
- `lkj_cholesky_eta`: This is a vector of length `n_class`, where each element corresponds to the LKJ prior parameter η_c corresponding to latent class c. For non-latent class models (i.e., the MVP), this will just be a vector with 1 element e.g. c(4) corresponds to: $\Omega \sim$ LKJ (4). For latent class models (i.e. the LC_MVP and the latent_trait models), the first element corresponds to the first latent class (for test accuracy applications this will be the NON-diseased class) and the second element corresponds to the second latent class (for test accuracy applications this will be the diseased class).
- `ub_corr`: An array of dimension `n_class` x `n_outcomes` x `n_outcomes`, which contains the upper-bounds for the correlations. Note that only the lower-triangular elements of each of the supplied `n_class` matrices are used, and the default is a matrix with lower-triangular part all equal to 1.
- `lb_corr`: An array of dimension `n_class` x `n_outcomes` x `n_outcomes`, which contains the lower-bounds for the correlations. Note that only the lower-triangular elements of each of the supplied `n_class` matrices are used, and the default is a matrix with lower-triangular part all equal to 0.
- `known_values_indicator`: An array of dimension `n_class` x `n_outcomes` x `n_outcomes`, which contains elements that are either 1 or 0, such that: if element (c, t_1, t_2) is 0, then correlation (c, t_1, t_2) is unknown and will be estimated, however if element (c, t_1, t_2) is 1, then we know correlation (c, t_1, t_2) a priori and hence it will be fixed - specifically it will be set equal to the corresponding value in `known_values` (see below). In other words, if any correlations are known a priori, they can be passed onto the model via this argument. Note that only the lower-triangular elements of each of the supplied `n_class` matrices are used, and the default is a matrix with lower-triangular part all equal to 0 (i.e., assumes no # correlations are known/fixed).
- `known_values`: An array of dimension `n_class` x `n_outcomes` x `n_outcomes`, which contains any known values for the correlations. In other words, if any correlations are known a priori, they can be passed onto the model via this argument. Note that only the lower-triangular elements of each of the supplied `n_class` matrices are used, and the default is a matrix with lower-triangular part all equal to 0 (note that these values are arbitrary since the elements in known_values_indicator are all equal to zero, so they will be ignored unless one or more elements in known_values_indicator is non-zero).

Arguments which are only relevant to latent class models (i.e. the LC_MVP and latent_trait models):

- `prev_prior_a`: Shape parameter 1 for prevalence beta prior. Only relevant for latent-class models (i.e. the LC_MVP or latent_trait models).
- `prev_prior_b`: Shape parameter 2 for prevalence beta prior. Only relevant for latent-class models (i.e. the LC_MVP or latent_trait models).

Arguments which are only relevant to the latent trait model (i.e., if `Model_type = "latent_trait"`) include:

- `LT_b_priors_shape`: A matrix of dimension `n_class` x `n_outcomes`, where each element corresponds to the prior Weibull shape parameter of the "b" parameters in the latent trait model - which are denoted as `LT_b`. The default is a matrix with with every value equal to 1.33. Please see LT_b_priors_scale below for a justification of this default choice.
- `LT_b_priors_scale`: A matrix of dimension `n_class` x `n_outcomes`, where each element corresponds to the prior Weibull scale parameter of the "b" parameters in the latent trait model - which are denoted as `LT_b`. The default is a matrix with every value equal to 1.25. Together with the default choice `LT_b_priors_shape` (please see description above), these priors correspond to the following Weibull priors: $b_{c,t} \sim$ Weibull $(1.33, 1.250)$. We chose these as default values because they are equivalent to setting truncated-LKJ (1.5) priors in the LC_MVP model, which are very weakly informative, especially if the dimension (i.e. number of outcomes/tests) is small.
- `LT_known_bs_indicator`: A matrix of dimension `n_class` x `n_outcomes`, which contains elements that are either 1 or 0, such that: if element (c, t) is 0, then the corresponding `LT_b` parameter is unknown and will be estimated; however, if element (c, t) is 1, then we know the corresponding `LT_b` parameter a priori and hence it will be fixed - specifically it will be set equal to the corresponding value in `LT_known_bs_values`. In other words, if any `LT_b` are known a priori, they can be passed onto the model via this argument.
- `LT_known_bs_values`: A matrix of dimension `n_class` x `n_outcomes`, which contains any known values for the `LT_b` parameters. In other words, if any `LT_b` are known a priori, they can be passed onto the model via this argument.

`Stan_data_list` List containing data for Stan models (only relevant if "Model_type" is set to "Stan"). The elements of the list should correspond to the variables defined in the "data" block of your Stan model.

`sample_nuisance` Whether or not to sample the high-dimensional nuisance/latent variable vector.

`n_chains_burnin` The total number of burn-in chains. The default is Min(8, n_cores), where n_cores is the number of cores on the CPU.

Methods

Public methods:

- `MVP_model$new()`
- `MVP_model$sample()`
- `MVP_model$summary()`
- `MVP_model$clone()`

Method `new()`**:** Create a new MVP model object

Usage:

```
MVP_model$new(
  Model_type,
  y,
  N,
  n_params_main,
  n_nuisance,
  init_lists_per_chain,
  n_chains_burnin,
  compile = TRUE,
  force_recompile = TRUE,
  model_args_list = NULL,
  Stan_data_list = NULL,
  sample_nuisance = NULL,
  Stan_model_file_path = NULL,
  Stan_cpp_user_header = NULL,
  Stan_cpp_flags = NULL,
  ...
)
```

Arguments:

`Model_type` Type of model ("MVP", "LC-MVP", etc.). See class documentation for details.

`y` The dataset. See class documentation for details.

`N` The sample size. See class documentation for details.

`n_params_main` Number of main parameters. See class documentation for details.

`n_nuisance` Number of nuisance parameters. See class documentation for details.

`init_lists_per_chain` List of initial values for each chain. See class documentation for details.

`n_chains_burnin` Number of chains used for burnin. See class documentation for details.

`compile` Compile the (possibly dummy if using built-in models) Stan model.

`force_recompile` Force-compile the (possibly dummy if using built-in models) Stan model.

`model_args_list` List of model arguments. See class documentation for details.

`Stan_data_list` List of Stan data (optional). See class documentation for details.

`sample_nuisance` Whether to sample nuisance parameters. See class documentation for details.

`Stan_model_file_path` The file path to the Stan model, only needed if `Model_type = "Stan"`.

`Stan_cpp_user_header` The file path to a user-supplied C++ .hpp file to be compiled together with the Stan model. This is optional and only needed if you want to use custom C++ functions in your Stan model, and is only relvant if `Model_type = "Stan"`.

`Stan_cpp_flags` User-supplied R list containing comma-separated values of compiler flags (e.g. CXX_FLAGS, etc) to be passed on to cmdstanr - the Stan model will then be compiled using these flags. This is optional and only needed if you want to use custom C++ functions in your Stan model. Only relevant if `Model_type = "Stan"`.

`...` Additional arguments passed to BayesMVP::initialise_model.

Returns: Returns self$init_object, an object generated from the "BayesMVP::initialise_model" function which contains information such as which model type (`Model_type`) to use.

Method `sample()`**:** Sample from the model

Usage:

```
MVP_model$sample(
  init_lists_per_chain = self$init_lists_per_chain,
  model_args_list = self$model_args_list,
  Stan_data_list = self$Stan_data_list,
  parallel_method = "RcppParallel",
  y = self$y,
  N = self$N,
  manual_tau = NULL,
  tau_if_manual = NULL,
  sample_nuisance = self$sample_nuisance,
  partitioned_HMC = FALSE,
  diffusion_HMC = FALSE,
  vect_type = NULL,
  Phi_type = "Phi",
  n_params_main = self$n_params_main,
  n_nuisance = self$n_nuisance,
  n_chains_burnin = self$n_chains_burnin,
  n_chains_sampling = NULL,
  n_superchains = NULL,
  seed = NULL,
  n_burnin = 500,
  n_iter = 1000,
  adapt_delta = 0.8,
  learning_rate = NULL,
  clip_iter = NULL,
  interval_width_main = NULL,
  interval_width_nuisance = NULL,
  force_autodiff = FALSE,
  force_PartialLog = FALSE,
  multi_attempts = TRUE,
  max_L = 1024,
  tau_mult = 1.6,
  metric_type_main = "Hessian",
  metric_shape_main = "diag",
  metric_type_nuisance = "Empirical",
  ratio_M_main = 0.25,
  ratio_M_us = 0.25,
  force_recompile = FALSE,
  ...
)
```

Arguments:

`init_lists_per_chain` List of initial values for each chain. See class documentation for details.

`model_args_list` List of model arguments. See class documentation for details.

`Stan_data_list` List of Stan data (optional). See class documentation for details.

`parallel_method` The method to use for parallelisation (multithreading) in C++. Default is "RcppParallel". Can be changed to "OpenMP", if available.

`y` The dataset. See class documentation for details.

`N` The sample size. See class documentation for details.

`manual_tau` If `FALSE`, then SNAPER-HMC will be used to adapt τ during the burnin phase. Otherwise if `TRUE`, τ will be fixed to the value given in the `tau_if_manual` argument.

`tau_if_manual` The HMC path length (τ) to use for the HMC sampling. This will be used for both the burnin and sampling phases. Note that this only works if `manual_tau = TRUE`. Otherwise, τ will be adapted using SNAPER-HMC. Also note that if only one value is given, then this value of tau will be used for both the main parameter sampling and the nuisance parameter sampling. If you want to specify separate path lengths for the main and nuisance parameters, provide a vector instead. E.g. `tau_if_manual = c(0.50, 2.0)` will mean that τ = 0.50 is used for the main parameters and τ = 2.0 is used for the nuisance parameters.

`sample_nuisance` Whether to sample nuisance parameters. See class documentation for details.

`partitioned_HMC` Whether to sample all parameters at once (note: wont use diffusion HMC) or whether to alternate between sampling the nuisance parameters and the main model parameters.

`diffusion_HMC` Whether to use diffusion-pathspace HMC (Beskos et al) to sample nuisance parameters. Default is TRUE.

`vect_type` The SIMD (single-nstruction, multiple-data) vectorisation type to use for math functions (such as log, exp, Phi, etc). The default is AVX-512 if available, then AVX2 if not, and if neither AVX-512 nor AVX2 are available (e.g., on ARM-based CPU's such as Apple systems), then BayesMVP will use the Stan C++ math library functions, and they will decide what vectorisation type to use. More vectorisation types will be supported in the future.

`Phi_type` Type of Phi() function implementation to use, where Phi() is the standard normal CDF. Can be either "Phi" (the default) or "Phi_approx()".

`n_params_main` Number of main parameters. See class documentation for details.

`n_nuisance` Number of nuisance parameters. See class documentation for details.

`n_chains_burnin` Number of chains used for burnin. See class documentation for details.

`n_chains_sampling` Number of chains used for sampling (i.e., post-burnin). Note that this must match the length of "init_lists_per_chain".

`n_superchains` Number of superchains for nested R-hat (nR-hat; see Margossian et al, 2023) computation.

`seed` Random MCMC seed.

`n_burnin` The number of burnin iterations. Default is 500.

`n_iter` The number of sampling (i.e., post-burnin) iterations. Default is 1000.

`adapt_delta` The Metropolis-Hastings target acceptance rate. Default is 0.80. If there are divergences, sometimes increasing this can help, at the cost of efficiency (since this will decrease the step-size (epsilon) and increase the number of leapfrog steps (L) per iteration).

`learning_rate` The ADAM learning rate (LR) for learning the appropriate HMC step-size (ϵ) and HMC path length ($\tau = L \cdot \epsilon$) during the burnin period. The default depends on the length of burnin chosen as follows: if `n_burnin` < 249, then LR=0.10, if it's between 250 and 500 then LR = 0.075, and if the burnin length is between 501 and 750 then LR=0.05, and finally if the burnin is >750 iterations then LR=0.025.

`clip_iter` The number of iterations to perform MALA (i.e., one leapfrog step) on (first clip_iter iterations) for the burnin period. The default depends on the length of the burnin period.

`interval_width_main` How often to update the metric (which is either empirical or Hessian-informed, and can be diagonal or dense) for the main parameters during the burnin period. The default used depends on the length of the burnin period.

`interval_width_nuisance` How often to update the metric (which is empirical and diagonal) for the nuisance parameters during the burnin period. The default used depends on the length of the burnin period.

`force_autodiff` Whether to use (force) autodiff instead of the built-in manual gradients. This is only relevant for the built-in models, not Stan models. The default is `FALSE`.

`force_PartialLog` Whether to use (force) use of gradients on the log-scale (for stability). Note that if `force_autodiff = TRUE` then this will automatically be set to `TRUE`, since autodiff is only available for partial-log-scale gradients. The default is `FALSE`.

`multi_attempts` Whether

`max_L` The maximum number of leapfrog steps. This is similar to `max_treedepth` in Stan. The default is 1024 (which is equivalent to the default `max_treedepth = 10` in Stan).

`tau_mult` The SNAPER-HMC algorithm multiplier to use when adjusting the path length during the burnin phase.

`metric_type_main` The type of metric to use for the main parameters, which is adapted during the burnin period. Can be either `"Hessian"` or `"Empirical"`, where the former uses second derivative information and the latter uses the SD of the posterior computed whilst sampling. The default is `"Hessian"` if `n_params_main < 250` and `"Empirical"` otherwise.

`metric_shape_main` The shape of the metric to use for the main parameters, which is adapted during the burnin period. Can be either `"diag"` or `"dense"`. The default is `"diag"` (i.e. a diagonal metric).

`metric_type_nuisance` The type of metric to use for the nuisance parameters, which is adapted during the burnin period. Can be "diag" or "unit".

`ratio_M_main` Ratio to use for the metric. Main parameters.

`ratio_M_us` Ratio to use for the metric. Nuisance parameters.

`force_recompile` Recompile the (possibly dummy if using built-in model) Stan model.

`...` Additional arguments passed to sampling.

Returns: Returns self invisibly, allowing for method chaining of this classes (MVP_model) methods. E.g.: model$sample(...)$summary(...).

Method `summary()`**:** Create and compute summary statistics, traces and model diagnostics.

Usage:

```
MVP_model$summary(
  compute_main_params = TRUE,
  compute_transformed_parameters = TRUE,
  compute_generated_quantities = TRUE,
  save_log_lik_trace = TRUE,
  save_nuisance_trace = FALSE,
  compute_nested_rhat = NULL,
  n_superchains = NULL,
  save_trace_tibbles = FALSE,
  ...
)
```

Arguments:

`compute_main_params` Whether to compute the main parameter summaries. Default is TRUE. Note that this excludes the high-dimensional nuisance parameter vector (for Stan models -

this should be defined as the FIRST parameter in the "parameters" block). For Stan models, this will be for the parameters defined the "parameters" block of the model. For built-in models (i.e. the MVP, LC_MVP, and latent_trait), this will compute summaries and traces for the coefficient vector (beta; for all 3 models), the correlation matrix/matrices (Omega; for the MVP and LC_MVP only), for the latent_trait latent effect coefficients (i.e. the "b" parameters - denoted "LT_b" - for latent_trait only), and finally for the disease prevalence ("p" or "prev"; for the LC_MVP and latent_trait only).

`compute_transformed_parameters` Whether to compute transformed parameter summaries. Default is TRUE. For Stan models, this will be for all of the parameters defined in the "transformed parameters" block, EXCEPT for the (transformed) nuisance parameters and log_lik (see "save_log_lik_trace" for more information on log_lik).

`compute_generated_quantities` Whether to compute the summaries for generated quantities. Default is TRUE. For Stan models this will exclude any log_lik defined in the "generated quantities" block - see "save_log_lik_trace".

`save_log_lik_trace` Whether to save the log-likelihood (log_lik) trace. Default is FALSE. For Stan models, this will only work if there is a "log_lik" parameter defined in the "transformed parameters" model block. For built-in models,

`save_nuisance_trace` Whether to save the nuisance trace. Default is FALSE.

`compute_nested_rhat` Whether to compute the nested rhat diagnostic (nR-hat) (Margossian et al, 2023). This is useful when running many (usually short) chains. Also see "n_superchains" argument.

`n_superchains` The number of superchains to use for the computation of nR-hat. Only relevant if `compute_nested_rhat = TRUE`.

`save_trace_tibbles` Whether to save the trace as tibble dataframes as well as 3D arrays. Default is FALSE.

`...` Any other arguments to be passed to BayesMVP::create_summary_and_traces.

Returns: Returns a new MVP_class_extract_and_plot object (from the "MVP_class_extract_and_plot" R6 class) for creating MCMC diagnostics and plots.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`set_pkg_example_path_and_wd` *set_pkg_example_path_and_wd*

Description

set_pkg_example_path_and_wd

Usage

```
set_pkg_example_path_and_wd()
```

Index