project-template Documentation

Release 0.9.66

Stan Development Team

Sep 23, 2020

Contents:

CHAPTER 1

Getting Started

1.1 Installation

1.1.1 Install package CmdStanPy

CmdStanPy is a pure-Python package which can be installed from PyPI

```
pip install --upgrade cmdstanpy
```

or from GitHub

```
pip install -e git+https://github.com/stan-dev/cmdstanpy#egg=cmdstanpy
```

To install CmdStanPy with all the optional packages (ujson; json processing, tqdm; progress bar)

```
pip install --upgrade cmdstanpy[all]
```

Note for PyStan users: PyStan and CmdStanPy should be installed in separate environments. If you already have PyStan installed, you should take care to install CmdStanPy in its own virtual environment.

User can install optional packages with pip with the CmdStanPy installation

```
pip install --upgrade cmdstanpy[all]
```

The optional packages are

- `ujson` which provides faster IO
- `tqdm` which displays a progress during sampling

To install these manually

```
pip install ujson
pip install tqdm
```

1.1.2 Install CmdStan

CmdStanPy requires a local install of CmdStan.

1.1.2.1 Prerequisites

CmdStanPy requires an installed C++ toolchain consisting of a modern C++ compiler and the GNU-Make utility.

1.1.2.2 Function `install_cmdstan`

CmdStanPy provides the function `install_cmdstan` which downloads CmdStan from GitHub and builds the CmdStan utilities. It can be can be called from within Python or from the command line. By default it installs the latest version of CmdStan into a directory named `.cmdstanpy` in your `$HOME` directory:

- From Python

```
import cmdstanpy
cmdstanpy.install_cmdstan()
```

- From the command line on Linux or MacOSX

```
install_cmdstan
ls -F ~/.cmdstanpy
```

- On Windows

```
python -m cmdstanpy.install_cmdstan
dir "%HOME%/.cmdstanpy"
```

The named arguments: *-d <directory>* and *-v <version>* can be used to override these defaults:

```
install_cmdstan -d my_local_cmdstan -v 2.20.0
ls -F my_local_cmdstan
```

Specifying CmdStan installation location

The default for the CmdStan installation location is a directory named `.cmdstanpy` in your `$HOME` directory.

If you have installed CmdStan in a different directory, then you can set the environment variable `CMDSTAN` to this location and it will be picked up by CmdStanPy:

```
export CMDSTAN='/path/to/cmdstan-2.20.0'
```

The CmdStanPy commands `cmdstan_path` and `set_cmdstan_path` get and set this environment variable:

```
from cmdstanpy import cmdstan_path, set_cmdstan_path

oldpath = cmdstan_path()
set_cmdstan_path(os.path.join('path','to','cmdstan'))
newpath = cmdstan_path()
```

Specifying a custom Make tool

To use custom `make`-tool use `set_make_env` function.

```
from cmdstanpy import set_make_env
set_make_env("mingw32-make.exe") # On Windows with mingw32-make
```

1.1.2.3 Installion via the command line

If you with to install CmdStan yourself, follow the instructions in the CmdStan User's Guide.

1.2 "Hello, World"

1.2.1 Bayesian estimation via Stan's HMC-NUTS sampler

To exercise the essential functions of CmdStanPy we show how to run Stan's HMC-NUTS sampler to estimate the posterior probability of the model parameters conditioned on the data. Do do this we use the example Stan model `bernoulli.stan` and corresponding dataset `bernoulli.data.json` which are distributed with CmdStan.

This is a simple model for binary data: given a set of N observations of i.i.d. binary data *y[1] ... y[N]*, it calculates the Bernoulli chance-of-success *theta*.

```
data {
   int<lower=0> N;
   int<lower=0,upper=1> y[N];
 }
 parameters {
   real<lower=0,upper=1> theta;
 }
 model {
   theta ~ beta(1,1);  // uniform prior on interval 0,1
   y ~ bernoulli(theta);
 }
```

The data file specifies the number of observations and their values.

```
{
 "N" : 10,
 "y" : [0,1,0,0,0,0,0,0,0,1]
}
```

1.2.1.1 Specify a Stan model

The: *CmdStanModel* class manages the Stan program and its corresponding compiled executable. It provides properties and functions to inspect the model code and filepaths. By default, the Stan program is compiled on instantiation.

```
import os
from cmdstanpy import cmdstan_path, CmdStanModel

bernoulli_stan = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.stan
↪')
bernoulli_model = CmdStanModel(stan_file=bernoulli_stan)
```

(continues on next page)

(continued from previous page)

```
bernoulli_model.name
bernoulli_model.stan_file
bernoulli_model.exe_file
bernoulli_model.code()
```

1.2.1.2 Run the HMC-NUTS sampler

The *CmdStanModel* method `sample` is used to do Bayesian inference over the model conditioned on data using using Hamiltonian Monte Carlo (HMC) sampling. It runs Stan's HMC-NUTS sampler on the model and data and returns a *CmdStanMCMC* object.

```
bernoulli_data = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.
↪data.json')
bern_fit = bernoulli_model.sample(data=bernoulli_data, output_dir='.')
```

By default, the `sample` command runs 4 sampler chains. This is a set of per-chain Stan CSV files The filenames follow the template '<model_name>-<YYYYMMDDHHMM>-<chain_id>' plus the file suffix '.csv'. There is also a correspondingly named file with suffix '.txt' which contains all messages written to the console. If the `output_dir` argument is omitted, the output files are written to a temporary directory which is deleted when the current Python session is terminated.

1.2.1.3 Access the sample

The *CmdStanMCMC* object stores the CmdStan config information and the names of the the per-chain output files. It manages and retrieves the sampler outputs as Python objects.

```
print(bern_fit)
```

The resulting set of draws produced by the sampler is lazily instantiated as a 3-D `numpy.ndarray` (i.e., a multi-dimensional array) over all draws from all chains arranged as draws X chains X columns. Instantiation happens the first time that any of the information in the posterior is accesed via properties: `draws`, `metric`, or `stepsize` are accessed. At this point the stan-csv output files are read into memory. For large files this may take several seconds; for the example dataset, this should take less than a second.

```
bern_fit.draws().shape
```

Python's index slicing operations can be used to access the information by chain. For example, to select all draws and all output columns from the first chain, we specify the chain index (2nd index dimension). As arrays indexing starts at 0, the index '0' corresponds to the first chain in the *CmdStanMCMC*:

```
chain_1 = bern_fit.draws()[:,0,:]
chain_1.shape       # (1000, 8)
chain_1[0]          # first draw:
                    # array([-7.99462  ,  0.578072 ,  0.955103 ,  2.       ,  7.
↪ ,
                    # 0.       ,  9.44788  ,  0.0934208])
```

To work with the draws from all chains for a parameter or quantity of interest in the model, use the `stan_variable` method to obtains a numpy.ndarray which contains the set of draws in the sample for the named Stan program variable by flattening the draws by chains into a single column:

```
bern_fit.stan_variable('theta')
```

1.2.1.4 Summarize or save the results

CmdStan is distributed with a posterior analysis utility stansummary that reads the outputs of all chains and computes summary statistics for all sampler and model parameters and quantities of interest. The *CmdStanMCMC* method `summary` runs this utility and returns summaries of the total joint log-probability density **lp__** plus all model parameters and quantities of interest in a pandas.DataFrame:

```
bern_fit.summary()
```

CmdStan is distributed with a second posterior analysis utility diagnose which analyzes the per-draw sampler parameters across all chains looking for potential problems which indicate that the sample isn't a representative sample from the posterior. The `diagnose` method runs this utility and prints the output to the console.

```
bern_fit.diagnose()
```

The `save_csvfiles` function moves the CmdStan csv output files to a specified directory.

```
bern_fit.save_csvfiles(dir='some/path')
```

CHAPTER 2

Stan Models in CmdStanPy

The: *CmdStanModel* class manages the Stan program and its corresponding compiled executable. It provides properties and functions to inspect the model code and filepaths. By default, the Stan program is compiled on instantiation.

```
import os
from cmdstanpy import cmdstan_path, CmdStanModel

bernoulli_stan = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.stan
↪')
bernoulli_model = CmdStanModel(stan_file=bernoulli_stan)
bernoulli_model.name
bernoulli_model.stan_file
bernoulli_model.exe_file
bernoulli_model.code()
```

A model object can be instantiated by specifying either the Stan program file path or the compiled executable file path or both. If both are specified, the constructor will check the timestamps on each and will only re-compile the program if the Stan file has a later timestamp which indicates that the program may have been edited.

2.1 Model compilation

Model compilation is carried out via the GNU Make build tool. The CmdStan `makefile` contains a set of general rules which specify the dependencies between the Stan program and the Stan platform components and low-level libraries. Optional behaviors can be specified by use of variables which are passed in to the `make` command as name, value pairs.

Model compilation is done in two steps:

- The `stanc` compiler translates the Stan program to C++.
- The C++ compiler compiles the generated code and links in the necessary supporting libraries.

Therefore, both the constructor and the `compile` method allow optional arguments `stanc_options` and `cpp_options` which specify options for each compilation step. Options are specified as a Python dictionary mapping compiler option names to appropriate values.

To use Stan's parallelization features, Stan programs must be compiled with the appropriate C++ compiler flags. If you are running GPU hardware and wish to use the OpenCL framework to speed up matrix operations, you must set the C++ compiler flag **STAN_OPENCL**. For high-level within-chain parallelization using the Stan language *reduce_sum* function, it's necessary to set the C++ compiler flag **STAN_THREADS**. While any value can be used, we recommend the value `True`.

For example, given Stan program named 'proc_parallel.stan', you can take advantage of both kinds of parallelization by specifying the compiler options when instantiating the model:

```
proc_parallel_model = CmdStanModel(
    stan_file='proc_parallel.stan',
    cpp_options={"STAN_THREADS": True, "STAN_OPENCL": True},
)
```

2.1.1 Specifying a custom Make tool

To use custom Make-tool use `set_make_env` function.

```
from cmdstanpy import set_make_env
set_make_env("mingw32-make.exe") # On Windows with mingw32-make
```

CHAPTER 3

MCMC Sampling

The *CmdStanModel* class method `sample` invokes Stan's adaptive HMC-NUTS sampler which uses the Hamiltonian Monte Carlo (HMC) algorithm and its adaptive variant the no-U-turn sampler (NUTS) to produce a set of draws from the posterior distribution of the model parameters conditioned on the data.

In order to evaluate the fit of the model to the data, it is necessary to run several Monte Carlo chains and compare the set of draws returned by each. By default, the `sample` command runs 4 sampler chains, i.e., CmdStanPy invokes CmdStan 4 times. CmdStanPy uses Python's `subprocess` and `multiprocessing` libraries to run these chains in separate processes. This processing can be done in parallel, up to the number of processor cores available.

3.1 NUTS-HMC sampler configuration

- `chains`: Number of sampler chains.
- `parallel_chains`: Number of processes to run in parallel.
- `seed`: The seed or list of per-chain seeds for the sampler's random number generator.
- `chain_ids`: The offset or list of per-chain offsets for the random number generator.
- `inits`: Specifies how the sampler initializes parameter values.
- `iter_warmup`: Number of warmup iterations for each chain.
- `iter_sampling`: Number of draws from the posterior for each chain.
- `save_warmup`: When `True`, sampler saves warmup draws as part of output csv file.
- `thin`: Period between saved samples.
- `max_treedepth`: Maximum depth of trees evaluated by NUTS sampler per iteration.
- `metric`: Specification of the mass matrix.
- `step_size`: Initial stepsize for HMC sampler.
- `adapt_engaged`: When `True`, tune stepsize and metric during warmup. The default is `True`.

- `adapt_delta`: Adaptation target Metropolis acceptance rate. The default value is 0.8. Increasing this value, which must be strictly less than 1, causes adaptation to use smaller step sizes. It improves the effective sample size, but may increase the time per iteration.
- `adapt_init_phase`: Iterations for initial phase of adaptation during which step size is adjusted so that the chain converges towards the typical set.
- `adapt_metric_window`: The second phase of adaptation tunes the metric and stepsize in a series of intervals. This parameter specifies the number of iterations used for the first tuning interval; window size increases for each subsequent interval.
- `adapt_step_size`: Number of iterations given over to adjusting the step size given the tuned metric during the final phase of adaptation.
- `fixed_param`: When `True`, call CmdStan with argument "algorithm=fixed_param".
- `data`: Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- `seed`: The seed for random number generator.
- `inits`: Specifies how the sampler initializes parameter values.
- `output_dir`: Name of the directory to which CmdStan output files are written.
- `save_diagnostics`: Whether or not to the CmdStan auxiliary output file. For the `sample` method, the diagnostics file contains sampler information for each draw together with the gradients on the unconstrained scale and log probabilities for all parameters in the model.

All of these arguments are optional; when unspecified, the CmdStan defaults will be used.

3.2 Example: fit model - sampler defaults

In this example we use the CmdStan example model bernoulli.stan and data file bernoulli.data.json.

The *CmdStanModel* class method `sample` returns a `CmdStanMCMC` object which provides properties to retrieve information about the sample, as well as methods to run CmdStan's summary and diagnostics tools.

Methods for information about the fit of the model to the data:

- `summary()` - Run CmdStan's stansummary utility on the sample.
- `diagnose()` - Run CmdStan's diagnose utility on the sample.
- `sampler_diagnostics()` - Returns the sampler parameters as a map from sampler parameter names to a numpy.ndarray of dimensions draws X chains X 1.

Methods for managing the sample:

- `save_csvfiles(dir_name)` - Move output csvfiles to specified directory.
- `chains` - Number of chains
- `num_draws` - Number of post-warmup draws (i.e., sampling iterations)
- `num_warmup_draws` - Number of warmup draws.
- `metric` - Per chain metric by the HMC sampler.
- `stepsize` - Per chain stepszie used by the HMC sampler.
- `sample` - A 3-D numpy.ndarray which contains all post-warmup draws across all chains arranged as (draws, chains, columns).

- `warmup` - A 3-D numpy.ndarray which contains all warmup draws across all chains arranged as (draws, chains, columns).

Methods for downstream analysis are:

- `stan_variable(var_name)` - Returns a numpy.ndarray which contains the set of draws in the sample for the named Stan program variable.
- `stan_variables()` - Return dictionary of all Stan program variables.

By default the sampler runs 4 chains, running as many chains in parallel as there are available processors as determined by Python's `multiprocessing.cpu_count()` function. For example, on a dual-processor machine with 4 virtual cores, all 4 chains will be run in parallel. Continuing this example, specifying `chains=6` will result in 4 chains being run in parallel, and as soon as 2 of them are finished, the remaining 2 chains will run. Specifying `chains=6, parallel_chains=6` will run all 6 chains in parallel.

```
import os
from cmdstanpy import cmdstan_path, CmdStanModel
bernoulli_stan = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.stan
↪')
bernoulli_data = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.
↪data.json')

# instantiate, compile bernoulli model
bernoulli_model = CmdStanModel(stan_file=bernoulli_stan)

# run the NUTS-HMC sampler
bern_fit = bernoulli_model.sample(data=bernoulli_data)
bern_fit.draws().shape
bern_fit.summary()
```

3.3 Example: high-level parallelization with reduce_sum

Stan provides high-level parallelization via multi-threading by use of the **reduce_sum** and **map_rect** functions in a Stan program. To use this feature, a Stan program must be compiled with the C++ compiler flag **STAN_THREADS** as described in the *Model compilation* section.

```
proc_parallel_model = CmdStanModel(
    stan_file='proc_parallel.stan',
    cpp_options={"STAN_THREADS": True}),
)
```

When running the sampler with this model, you must explicitly specify the number of threads to use via `sample` method argument **threads_per_chain**. For example, to run 4 chains multi-threaded using 4 threads per chain:

```
proc_parallel_fit = proc_parallel_model.sample(data=proc_data,
    chains=4, threads_per_chain=4)
```

By default, the number of parallel chains will be equal to the number of available cores on your machine, which may adversely affect overall performance. For example, on a machine with Intel's dual processor hardware, i.e, 4 virtual cores, the above configuration will use 16 threads. To limit this, specify the **parallel_chains** option so that the maximum number of threads used will be **parallel_chains** X **threads_per_chain**

```
proc_parallel_fit = proc_parallel_model.sample(data=proc_data,
    chains=4, parallel_chains=1, threads_per_chain=4)
```

3.4 Example: generate data - *fixed_param=True*

The Stan programming language can be used to write Stan programs which generate simulated data for a set of known parameter values by calling Stan's RNG functions. Such programs don't need to declare parameters or model blocks because all computation is done in the generated quantities block.

For example, the Stan program bernoulli.stan can be used to generate a dataset of simulated data, where each row in the dataset consists of *N* draws from a Bernoulli distribution given probability *theta*:

```
transformed data {
  int<lower=0> N = 10;
  real<lower=0,upper=1> theta = 0.35;
}
generated quantities {
  int y_sim[N];
  for (n in 1:N)
    y_sim[n] = bernoulli_rng(theta);
}
```

This program doesn't contain parameters or a model block, therefore we run the sampler without ding any MCMC estimation by specifying `fixed_param=True`. When `fixed_param=True`, the `sample` method only runs 1 chain. The sampler runs without updating the Markov Chain, thus the values of all parameters and transformed parameters are constant across all draws and only those values in the generated quantities block that are produced by RNG functions may change.

```
import os
from cmdstanpy import CmdStanModel
datagen_stan = os.path.join('..', '..', 'test', 'data', 'bernoulli_datagen.stan')
datagen_model = CmdStanModel(stan_file=datagen_stan)
sim_data = datagen_model.sample(fixed_param=True)
sim_data.summary()
```

Each draw contains variable *y_sim*, a vector of *N* binary outcomes. From this, we can compute the probability of new data given an estimate of parameter *theta* - the chance of success of a single Bernoulli trial. By plotting the histogram of the distribution of total number of successes in *N* trials shows the **posterior predictive distribution** of *theta*.

```
# extract int array `y_sim` from the sampler output
y_sims = sim_data.stan_variable(name='y_sim')
y_sims.shape

# each draw has 10 replicates of estimated parameter 'theta'
y_sums = y_sims.sum(axis=1)
# plot total number of successes per draw
import pandas as pd
y_sums_pd = pd.DataFrame(data=y_sums)
y_sums_pd.plot.hist(range(0,datagen_data['N']+1))
```

CHAPTER 4

Run Generated Quantities

The generated quantities block computes *quantities of interest* (QOIs) based on the data, transformed data, parameters, and transformed parameters. It can be used to:

- generate simulated data for model testing by forward sampling
- generate predictions for new data
- calculate posterior event probabilities, including multiple comparisons, sign tests, etc.
- calculating posterior expectations
- transform parameters for reporting
- apply full Bayesian decision theory
- calculate log likelihoods, deviances, etc. for model comparison

The *CmdStanModel* class `generate_quantities` method is useful once you have successfully fit a model to your data and have a valid sample from the posterior and a version of the original model where the generated quantities block contains the necessary statements to compute additional quantities of interest.

By running the `generate_quantities` method on the new model with a sample generated by the existing model, the sampler uses the per-draw parameter estimates from the sample to compute the generated quantities block of the new model.

The `generate_quantities` method returns a `CmdStanGQ` object which provides properties to retrieve information about the sample:

- `chains`
- `column_names`
- `generated_quantities`
- `generated_quantities_pd`
- `sample_plus_quantities`
- `save_csvfiles()`

The `sample_plus_quantities` combines the existing sample and new quantities of interest into a pandas DataFrame object which can be used for downstream analysis and visualization. In this way you add more columns of information to an existing sample.

4.1 Configuration

- `mcmc_sample` - either a `CmdStanMCMC` object or a list of stan-csv files
- `data`: Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- `seed`: The seed for random number generator.
- `gq_output_dir`: A path or file name which will be used as the basename for the CmdStan output files.

4.2 Example: add posterior predictive checks to `bernoulli.stan`

In this example we use the CmdStan example model bernoulli.stan and data file bernoulli.data.json as our existing model and data. We create the program bernoulli_ppc.stan by adding a `generated quantities` block to bernoulli.stan which generates a new data vector `y_rep` using the current estimate of theta.

```
generated quantities {
  int y_sim[N];
  real<lower=0,upper=1> theta_rep;
  for (n in 1:N)
    y_sim[n] = bernoulli_rng(theta);
  theta_rep = sum(y) / N;
}
```

The first step is to fit model `bernoulli` to the data:

```
import os
from cmdstanpy import CmdStanModel, cmdstan_path

bernoulli_dir = os.path.join(cmdstan_path(), 'examples', 'bernoulli')
bernoulli_path = os.path.join(bernoulli_dir, 'bernoulli.stan')
bernoulli_data = os.path.join(bernoulli_dir, 'bernoulli.data.json')

# instantiate, compile bernoulli model
bernoulli_model = CmdStanModel(stan_file=bernoulli_path)

# fit the model to the data
bern_fit = bernoulli_model.sample(data=bernoulli_data)
```

Then we compile the model `bernoulli_ppc` and use the fit parameter estimates to generate quantities of interest:

```
bernoulli_ppc_model = CmdStanModel(stan_file='bernoulli_ppc.stan')
bernoulli_ppc_model.compile()
new_quantities = bernoulli_ppc_model.generate_quantities(data=bern_data, mcmc_
↪sample=bern_fit)
```

The `generate_quantities` method returns a `CmdStanGQ` object which contains the values for all variables in the generated quantitites block of the program `bernoulli_ppc.stan`. Unlike the output from the `sample` method, it doesn't contain any information on the joint log probability density, sampler state, or parameters or transformed parameter values.

```
new_quantities.column_names
new_quantities.generated_quantities.shape
for i in range(len(new_quantities.column_names)):
    print(new_quantities.generated_quantities[:,i].mean())
```

The method `sample_plus_quantities` returns a pandas DataFrame which combines the input drawset with the generated quantities.

```
sample_plus = new_quantities.sample_plus_quantities
print(sample_plus.shape)
print(sample_plus.columns)
```

CHAPTER 5

Maximum Likelihood Estimation

Stan provides optimization algorithms which find modes of the density specified by a Stan program. Three different algorithms are available: a Newton optimizer, and two related quasi-Newton algorithms, BFGS and L-BFGS. The L-BFGS algorithm is the default optimizer. Newton's method is the least efficient of the three, but has the advantage of setting its own stepsize.

5.1 Optimize configuration

- `algorithm`: Algorithm to use. One of: "BFGS", "LBFGS", "Newton".
- `init_alpha`: Line search step size for first iteration.
- `iter`: Total number of iterations.
- `data`: Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- `seed`: The seed for random number generator.
- `inits`: Specifies how the sampler initializes parameter values.
- `output_dir`: Name of the directory to which CmdStan output files are written.
- `save_diagnostics`: Whether or not to the CmdStan auxiliary output file. For the `sample` method, the diagnostics file contains sampler information for each draw together with the gradients on the unconstrained scale and log probabilities for all parameters in the model.

All of these arguments are optional; when unspecified, the CmdStan defaults will be used.

5.2 Example: estamate MLE for model `bernoulli.stan` by optimization

In this example we use the CmdStan example model bernoulli.stan and data file bernoulli.data.json.

The *CmdStanModel* class method `optimize` returns a `CmdStanMLE` object which provides properties to retrieve the estimate of the penalized maximum likelihood estaimate of all model parameters:

- `column_names`
- `optimized_params_dict`
- `optimized_params_np`
- `optimized_params_pd`

In the following example, we instantiate a model and do optimization using the default CmdStan settings:

```
import os
from cmdstanpy.model import CmdStanModel
from cmdstanpy.utils import cmdstan_path

# instantiate, compile bernoulli model
bernoulli_path = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.stan
↪')
bernoulli_model = CmdStanModel(stan_file=bernoulli_path)

# run CmdStan's optimize method, returns object `CmdStanMLE`
bern_data = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.data.json
↪')
bern_mle = bernoulli_model.optimize(data=bernoulli_data)
print(bern_mle.column_names)
print(bern_mle.optimized_params_dict)
```

5.3 References

- Stan manual: https://mc-stan.org/docs/reference-manual/optimization-algorithms-chapter.html

CHAPTER 6

Variational Inference

Variational inference is a scalable technique for approximate Bayesian inference. In the Stan ecosystem, the terms "VI" and "VB" ("variational Bayes") are used synonymously.

Stan implements an automatic variational inference algorithm, called Automatic Differentiation Variational Inference (ADVI) which searches over a family of simple densities to find the best approximate posterior density. ADVI produces an estimate of the parameter means together with a sample from the approximate posterior density.

ADVI approximates the variational objective function, the evidence lower bound or ELBO, using stochastic gradient ascent. The algorithm ascends these gradients using an adaptive stepsize sequence that has one parameter `eta` which is adjusted during warmup. The number of draws used to approximate the ELBO is denoted by `elbo_samples`. ADVI heuristically determines a rolling window over which it computes the average and the median change of the ELBO. When this change falls below a threshold, denoted by `tol_rel_obj`, the algorithm is considered to have converged.

6.1 ADVI configuration

- `algorithm`: Algorithm to use. One of: "meanfield", "fullrank".
- `iter`: Maximum number of ADVI iterations.
- `grad_samples`: Number of MC draws for computing the gradient.
- `elbo_samples`: Number of MC draws for estimate of ELBO.
- `eta`: Stepsize scaling parameter.
- `adapt_iter`: Number of iterations for eta adaptation.
- `tol_rel_obj`: Relative tolerance parameter for convergence.
- `eval_elbo`: Number of interactions between ELBO evaluations.
- `output_samples`: Number of approximate posterior output draws to save.
- `data`: Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.

- `seed`: The seed for random number generator.
- `inits`: Specifies how the sampler initializes parameter values.
- `output_dir`: Name of the directory to which CmdStan output files are written.
- `save_diagnostics`: Whether or not to the CmdStan auxiliary output file. For the `sample` method, the diagnostics file contains sampler information for each draw together with the gradients on the unconstrained scale and log probabilities for all parameters in the model.

All of these arguments are optional; when unspecified, the CmdStan defaults will be used.

6.2 Example: variational inference for model `bernoulli.stan`

In this example we use the CmdStan example model bernoulli.stan and data file bernoulli.data.json.

The *CmdStanModel* class method `variational` returns a `CmdStanVB` object which provides properties to retrieve the estimate of the approximate posterior mean of all model parameters, and the returned set of draws from this approximate posterior (if any):

- `column_names`
- `variational_params_dict`
- `variational_params_np`
- `variational_params_pd`
- `variational_sample`
- `save_csvfiles()`

In the following example, we instantiate a model and run variational inference using the default CmdStan settings:

```
import os
from cmdstanpy.model import CmdStanModel
from cmdstanpy.utils import cmdstan_path

# instantiate, compile bernoulli model
bernoulli_path = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.stan
↪')
bernoulli_model = CmdStanModel(stan_file=bernoulli_path)

# run CmdStan's variational inference method, returns object `CmdStanVB`
bern_data = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.data.json
↪')
bern_vb = bernoulli_model.variational(data=bern_data)
print(bern_vb.column_names)
print(bern_vb.variational_params_dict)
bern_vb.variational_sample.shape
```

These estimates are only valid if the algorithm has converged to a good approximation. When the algorithm fails to do so, the `variational` method will throw a `RuntimeError`.

```
fail_stan = os.path.join(datafiles_path, 'variational', 'eta_should_fail.stan')
fail_model = CmdStanModel(stan_file=fail_stan)
model.compile()
vb = model.variational()
```

6.3 References

- Paper: [Kucukelbir et al](http://arxiv.org/abs/1506.03431)
- Stan manual: https://mc-stan.org/docs/reference-manual/vi-algorithms-chapter.html

CHAPTER 7

Under the Hood

Under the hood, CmdStanPy uses the CmdStan command line interface to compile and fit a model to data. The function `cmdstan_path` returns the path to the local CmdStan installation. See the installation section for more details on installing CmdStan.

7.1 File Handling

CmdStan is file-based interface, therefore CmdStanPy maintains the necessary files for all models, data, and inference method results. CmdStanPy uses the Python library `tempfile` module to create a temporary directory where all input and output files are written and which is deleted when the Python session is terminated.

7.1.1 Input Data

When the input data for the `CmdStanModel` inference methods is supplied as a Python dictionary, this data is written to disk as the corresponding JSON object.

7.1.2 Output Files

Output filenames are composed of the model name, a timestamp in the form 'YYYYMMDDhhmm' and the chain id, plus the corresponding filetype suffix, either '.csv' for the CmdStan output or '.txt' for the console messages, e.g. *bernoulli-201912081451-1.csv*. Output files written to the temporary directory contain an additional 8-character random string, e.g. *bernoulli-201912081451-1-5nm6as7u.csv*.

When the `output_dir` argument to the `CmdStanModel` inference methods is given, output files are written to the specified directory, otherwise they are written to the session-specific output directory. All fitted model objects, i.e. `CmdStanMCMC`, `CmdStanVB`, `CmdStanMLE`, and `CmdStanGQ`, have method `save_csvfiles` which moves the output files to a specified directory.

CHAPTER 8

API Reference

8.1 Classes

8.1.1 CmdStanModel

class cmdstanpy.**CmdStanModel**(*model_name: str = None*, *stan_file: str = None*, *exe_file: str = None*, *compile: bool = True*, *stanc_options: Dict = None*, *cpp_options: Dict = None*, *logger: logging.Logger = None*)

Stan model.

- Stores pathnames to Stan program, compiled executable, and collection of compiler options.
- Provides functions to compile the model and perform inference on the model given data.
- By default, compiles model on instantiation - override with argument `compile=False`
- By default, property `name` corresponds to basename of the Stan program or exe file - override with argument `model_name=<name>`.

code() → str

Return Stan program as a string.

compile(*force: bool = False*, *stanc_options: Dict = None*, *cpp_options: Dict = None*, *override_options: bool = False*) → None

Compile the given Stan program file. Translates the Stan code to C++, then calls the C++ compiler.

By default, this function compares the timestamps on the source and executable files; if the executable is newer than the source file, it will not recompile the file, unless argument `force` is `True`.

Parameters

- **force** – When `True`, always compile, even if the executable file is newer than the source file. Used for Stan models which have `#include` directives in order to force recompilation when changes are made to the included files.
- **compiler_options** – Options for stanc and C++ compilers.

- **override_options** – When `True`, override existing option. When `False`, add/replace existing options. Default is `False`.

cpp_options
Options to c++ compilers.

exe_file
Full path to Stan exe file.

generate_quantities(*data: Union[Dict, str] = None*, *mcmc_sample: Union[cmdstanpy.stanfit.CmdStanMCMC, List[str]] = None*, *seed: int = None*, *gq_output_dir: str = None*) → cmdstanpy.stanfit.CmdStanGQ
Run CmdStan's generate_quantities method which runs the generated quantities block of a model given an existing sample.

This function takes a CmdStanMCMC object and the dataset used to generate that sample and calls to the CmdStan `generate_quantities` method to generate additional quantities of interest.

The `CmdStanGQ` object records the command, the return code, and the paths to the generate method output csv and console files. The output files are written either to a specified output directory or to a temporary directory which is deleted upon session exit.

Output files are either written to a temporary directory or to the specified output directory. Output filenames correspond to the template '<model_name>-<YYYYMMDDHHMM>-<chain_id>' plus the file suffix which is either '.csv' for the CmdStan output or '.txt' for the console messages, e.g. 'bernoulli-201912081451-1.csv'. Output files written to the temporary directory contain an additional 8-character random string, e.g. 'bernoulli-201912081451-1-5nm6as7u.csv'.

Parameters

- **data** – Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- **mcmc_sample** – Can be either a `CmdStanMCMC` object returned by the `sample` method or a list of stan-csv files generated by fitting the model to the data using any Stan interface.
- **seed** – The seed for random number generator. Must be an integer between 0 and 2^32 - 1. If unspecified, `numpy.random.RandomState()` is used to generate a seed which will be used for all chains. *NOTE: Specifying the seed will guarantee the same result for multiple invocations of this method with the same inputs. However this will not reproduce results from the sample method given the same inputs because the RNG will be in a different state.*
- **gq_output_dir** – Name of the directory in which the CmdStan output files are saved. If unspecified, files will be written to a temporary directory which is deleted upon session exit.

Returns CmdStanGQ object

name
Model name used in output filename templates. Default is basename of Stan program or exe file, unless specified in call to constructor via argument *model_name*.

optimize(*data: Union[Dict, str] = None*, *seed: int = None*, *inits: Union[Dict, float, str] = None*, *output_dir: str = None*, *algorithm: str = None*, *init_alpha: float = None*, *iter: int = None*) → cmdstanpy.stanfit.CmdStanMLE
Run the specified CmdStan optimize algorithm to produce a penalized maximum likelihood estimate of the model parameters.

This function validates the specified configuration, composes a call to the CmdStan `optimize` method and spawns one subprocess to run the optimizer and waits for it to run to completion. Unspecified arguments are not included in the call to CmdStan, i.e., those arguments will have CmdStan default values.

The `CmdStanMLE` object records the command, the return code, and the paths to the optimize method output csv and console files. The output files are written either to a specified output directory or to a temporary directory which is deleted upon session exit.

Output files are either written to a temporary directory or to the specified output directory. Ouput filenames correspond to the template '<model_name>-<YYYYMMDDHHMM>-<chain_id>' plus the file suffix which is either '.csv' for the CmdStan output or '.txt' for the console messages, e.g. 'bernoulli-201912081451-1.csv'. Output files written to the temporary directory contain an additional 8-character random string, e.g. 'bernoulli-201912081451-1-5nm6as7u.csv'.

Parameters

- **data** – Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- **seed** – The seed for random number generator. Must be an integer between 0 and 2^32 - 1. If unspecified, `numpy.random.RandomState()` is used to generate a seed.
- **inits** – Specifies how the sampler initializes parameter values. Initialization is either uniform random on a range centered on 0, exactly 0, or a dictionary or file of initial values for some or all parameters in the model. The default initialization behavior will initialize all parameter values on range [-2, 2] on the *unconstrained* support. If the expected parameter values are too far from this range, this option may improve estimation. The following value types are allowed:
  - Single number, n > 0 - initialization range is [-n, n].
  - 0 - all parameters are initialized to 0.
  - dictionary - pairs parameter name : initial value.
  - string - pathname to a JSON or Rdump data file.
- **output_dir** – Name of the directory to which CmdStan output files are written. If unspecified, output files will be written to a temporary directory which is deleted upon session exit.
- **algorithm** – Algorithm to use. One of: 'BFGS', 'LBFGS', 'Newton'
- **init_alpha** – Line search step size for first iteration
- **iter** – Total number of iterations

Returns CmdStanMLE object

sample (*data: Union[Dict, str] = None, chains: Optional[int] = None, parallel_chains: Optional[int] = None, threads_per_chain: Optional[int] = None, seed: Union[int, List[int]] = None, chain_ids: Union[int, List[int]] = None, inits: Union[Dict, float, str, List[str]] = None, iter_warmup: int = None, iter_sampling: int = None, save_warmup: bool = False, thin: int = None, max_treedepth: float = None, metric: Union[str, List[str]] = None, step_size: Union[float, List[float]] = None, adapt_engaged: bool = True, adapt_delta: float = None, adapt_init_phase: int = None, adapt_metric_window: int = None, adapt_step_size: int = None, fixed_param: bool = False, output_dir: str = None, save_diagnostics: bool = False, show_progress: Union[bool, str] = False, validate_csv: bool = True*) → cmdstanpy.stanfit.CmdStanMCMC

Run or more chains of the NUTS sampler to produce a set of draws from the posterior distribution of a model conditioned on some data.

This function validates the specified configuration, composes a call to the CmdStan `sample` method and spawns one subprocess per chain to run the sampler and waits for all chains to run to completion.

Unspecified arguments are not included in the call to CmdStan, i.e., those arguments will have CmdStan default values.

For each chain, the `CmdStanMCMC` object records the command, the return code, the sampler output file paths, and the corresponding console outputs, if any. The output files are written either to a specified output directory or to a temporary directory which is deleted upon session exit.

Output files are either written to a temporary directory or to the specified output directory. Ouput filenames correspond to the template '<model_name>-<YYYYMMDDHHMM>-<chain_id>' plus the file suffix which is either '.csv' for the CmdStan output or '.txt' for the console messages, e.g. 'bernoulli-201912081451-1.csv'. Output files written to the temporary directory contain an additional 8-character random string, e.g. 'bernoulli-201912081451-1-5nm6as7u.csv'.

Parameters

- **data** – Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- **chains** – Number of sampler chains, must be a positive integer.
- **parallel_chains** – Number of processes to run in parallel. Must be a positive integer. Defaults to `multiprocessing.cpu_count()`.
- **threads_per_chain** – The number of threads to use in parallelized sections within an MCMC chain (e.g., when using the Stan functions `reduce_sum()` or `map_rect()`). This will only have an effect if the model was compiled with threading support. The total number of threads used will be `parallel_chains * threads_per_chain`.
- **seed** – The seed for random number generator. Must be an integer between 0 and 2^32 - 1. If unspecified, `numpy.random.RandomState()` is used to generate a seed which will be used for all chains. When the same seed is used across all chains, the chain-id is used to advance the RNG to avoid dependent samples.
- **chain_ids** – The offset for the random number generator, either an integer or a list of unique per-chain offsets. If unspecified, chain ids are numbered sequentially starting from 1.
- **inits** – Specifies how the sampler initializes parameter values. Initialization is either uniform random on a range centered on 0, exactly 0, or a dictionary or file of initial values for some or all parameters in the model. The default initialization behavior will initialize all parameter values on range [-2, 2] on the *unconstrained* support. If the expected parameter values are too far from this range, this option may improve adaptation. The following value types are allowed:
  - Single number n > 0 - initialization range is [-n, n].
  - 0 - all parameters are initialized to 0.
  - dictionary - pairs parameter name : initial value.
  - string - pathname to a JSON or Rdump data file.
  - list of strings - per-chain pathname to data file.
- **iter_warmup** – Number of warmup iterations for each chain.
- **iter_sampling** – Number of draws from the posterior for each chain.
- **save_warmup** – When `True`, sampler saves warmup draws as part of the Stan csv output file.
- **thin** – Period between saved samples.
- **max_treedepth** – Maximum depth of trees evaluated by NUTS sampler per iteration.

- **metric** – Specification of the mass matrix, either as a vector consisting of the diagonal elements of the covariance matrix ('diag' or 'diag_e') or the full covariance matrix ('dense' or 'dense_e').

  If the value of the metric argument is a string other than 'diag', 'diag_e', 'dense', or 'dense_e', it must be a valid filepath to a JSON or Rdump file which contains an entry 'inv_metric' whose value is either the diagonal vector or the full covariance matrix.

  If the value of the metric argument is a list of paths, its length must match the number of chains and all paths must be unique.

- **step_size** – Initial stepsize for HMC sampler. The value is either a single number or a list of numbers which will be used as the global or per-chain initial step size, respectively. The length of the list of step sizes must match the number of chains.

- **adapt_engaged** – When True, adapt stepsize and metric.

- **adapt_delta** – Adaptation target Metropolis acceptance rate. The default value is 0.8. Increasing this value, which must be strictly less than 1, causes adaptation to use smaller step sizes which improves the effective sample size, but may increase the time per iteration.

- **adapt_init_phase** – Iterations for initial phase of adaptation during which step size is adjusted so that the chain converges towards the typical set.

- **adapt_metric_window** – The second phase of adaptation tunes the metric and step-size in a series of intervals. This parameter specifies the number of iterations used for the first tuning interval; window size increases for each subsequent interval.

- **adapt_step_size** – Number of iterations given over to adjusting the step size given the tuned metric during the final phase of adaptation.

- **fixed_param** – When `True`, call CmdStan with argument `algorithm=fixed_param` which runs the sampler without updating the Markov Chain, thus the values of all parameters and transformed parameters are constant across all draws and only those values in the generated quantities block that are produced by RNG functions may change. This provides a way to use Stan programs to generate simulated data via the generated quantities block. This option must be used when the parameters block is empty. Default value is `False`.

- **output_dir** – Name of the directory to which CmdStan output files are written. If unspecified, output files will be written to a temporary directory which is deleted upon session exit.

- **save_diagnostics** – Whether or not to save diagnostics. If True, csv output files are written to an output file with filename template '<model_name>-<YYYYMMDDHHMM>-diagnostic-<chain_id>', e.g. 'bernoulli-201912081451-diagnostic-1.csv'.

- **show_progress** – Use tqdm progress bar to show sampling progress. If show_progress=='notebook' use tqdm_notebook (needs nodejs for jupyter).

- **validate_csv** – If `False`, skip scan of sample csv output file. When sample is large or disk i/o is slow, will speed up processing. Default is `True` - sample csv files are scanned for completeness and consistency.

Returns CmdStanMCMC object

stan_file
Full path to Stan program file.

stanc_options
Options to stanc compilers.

variational(*data: Union[Dict, str] = None*, *seed: int = None*, *inits: float = None*, *output_dir: str = None*, *save_diagnostics: bool = False*, *algorithm: str = None*, *iter: int = None*, *grad_samples: int = None*, *elbo_samples: int = None*, *eta: numbers.Real = None*, *adapt_engaged: bool = True*, *adapt_iter: int = None*, *tol_rel_obj: numbers.Real = None*, *eval_elbo: int = None*, *output_samples: int = None*, *require_converged: bool = True*) → cmdstanpy.stanfit.CmdStanVB

Run CmdStan's variational inference algorithm to approximate the posterior distribution of the model conditioned on the data.

This function validates the specified configuration, composes a call to the CmdStan `variational` method and spawns one subprocess to run the optimizer and waits for it to run to completion. Unspecified arguments are not included in the call to CmdStan, i.e., those arguments will have CmdStan default values.

The `CmdStanVB` object records the command, the return code, and the paths to the variational method output csv and console files. The output files are written either to a specified output directory or to a temporary directory which is deleted upon session exit.

Output files are either written to a temporary directory or to the specified output directory. Output filenames correspond to the template '<model_name>-<YYYYMMDDHHMM>-<chain_id>' plus the file suffix which is either '.csv' for the CmdStan output or '.txt' for the console messages, e.g. 'bernoulli-201912081451-1.csv'. Output files written to the temporary directory contain an additional 8-character random string, e.g. 'bernoulli-201912081451-1-5nm6as7u.csv'.

Parameters

- **data** – Values for all data variables in the model, specified either as a dictionary with entries matching the data variables, or as the path of a data file in JSON or Rdump format.
- **seed** – The seed for random number generator. Must be an integer between 0 and 2^32 - 1. If unspecified, `numpy.random.RandomState()` is used to generate a seed which will be used for all chains.
- **inits** – Specifies how the sampler initializes parameter values. Initialization is uniform random on a range centered on 0 with default range of 2. Specifying a single number n > 0 changes the initialization range to [-n, n].
- **output_dir** – Name of the directory to which CmdStan output files are written. If unspecified, output files will be written to a temporary directory which is deleted upon session exit.
- **save_diagnostics** – Whether or not to save diagnostics. If True, csv output files are written to an output file with filename template '<model_name>-<YYYYMMDDHHMM>-diagnostic-<chain_id>', e.g. 'bernoulli-201912081451-diagnostic-1.csv'.
- **algorithm** – Algorithm to use. One of: 'meanfield', 'fullrank'.
- **iter** – Maximum number of ADVI iterations.
- **grad_samples** – Number of MC draws for computing the gradient.
- **elbo_samples** – Number of MC draws for estimate of ELBO.
- **eta** – Stepsize scaling parameter.
- **adapt_engaged** – Whether eta adaptation is engaged.
- **adapt_iter** – Number of iterations for eta adaptation.
- **tol_rel_obj** – Relative tolerance parameter for convergence.
- **eval_elbo** – Number of iterations between ELBO evaluations.
- **output_samples** – Number of approximate posterior output draws to save.

- **require_converged** – Whether or not to raise an error if stan reports that "The algorithm may not have converged".

Returns CmdStanVB object

8.1.2 CmdStanMCMC

class cmdstanpy.**CmdStanMCMC**(*runset: cmdstanpy.stanfit.RunSet*, *validate_csv: bool = True*, *logger: logging.Logger = None*)

Container for outputs from CmdStan sampler run.

chain_ids
Chain ids.

chains
Number of chains.

column_names
all sampler and model parameters and quantities of interest

Type Names of all per-draw outputs

diagnose() → str
Run cmdstan/bin/diagnose over all output csv files. Returns output of diagnose (stdout/stderr).

The diagnose utility reads the outputs of all chains and checks for the following potential problems:

- Transitions that hit the maximum treedepth
- Divergent transitions
- Low E-BFMI values (sampler transitions HMC potential energy)
- Low effective sample sizes
- High R-hat values

draws(*inc_warmup: bool = False*) → numpy.ndarray
A 3-D numpy ndarray which contains all draws, from both warmup and sampling iterations, arranged as (draws, chains, columns) and stored column major, so that the values for each parameter are contiguous in memory, likewise all draws from a chain are contiguous.

Parameters **inc_warmup** – When `True` and the warmup draws are present in the output, i.e., the sampler was run with `save_warmup=True`, then the warmup draws are included. Default value is `False`.

draws_pd(*params: List[str] = None*, *inc_warmup: bool = False*) → pandas.core.frame.DataFrame
Returns the assembled draws as a pandas DataFrame consisting of one column per parameter and one row per draw.

Parameters

- **params** – list of model parameter names.
- **inc_warmup** – When `True` and the warmup draws are present in the output, i.e., the sampler was run with `save_warmup=True`, then the warmup draws are included. Default value is `False`.

metric
Metric used by sampler for each chain. When sampler algorithm 'fixed_param' is specified, metric is None.

metric_type
Metric type used for adaptation, either 'diag_e' or 'dense_e'. When sampler algorithm 'fixed_param' is specified, metric_type is None.

num_draws
Number of draws per chain.

sample
Deprecated - use method "draws()" instead.

sampler_diagnostics() → Dict
Returns the sampler diagnostics as a map from column name to draws X chains X 1 ndarray.

save_csvfiles(*dir: str = None*) → None
Move output csvfiles to specified directory. If files were written to the temporary session directory, clean filename. E.g., save 'bernoulli-201912081451-1-5nm6as7u.csv' as 'bernoulli-201912081451-1.csv'.

Parameters **dir** – directory path

stan_variable(*name: str*) → pandas.core.frame.DataFrame
Return a new DataFrame which contains the set of post-warmup draws for the named Stan program variable. Flattens the chains. Underlyingly draws are in chain order, i.e., for a sample consisting of N chains of M draws each, the first M array elements are from chain 1, the next M are from chain 2, and the last M elements are from chain N.

- If the variable is a scalar variable, the shape of the DataFrame is (draws X chains, 1).
- If the variable is a vector, the shape of the DataFrame is (draws X chains, len(vector))
- If the variable is a matrix, the shape of the DataFrame is (draws X chains, size(dim 1) X size(dim 2))
- If the variable is an array with N dimensions, the shape of the DataFrame is (draws X chains, size(dim 1) X . . . X size(dim N))

Parameters **name** – variable name

stan_variable_dims
Dict mapping Stan program variable names to variable dimensions. Scalar types have int value '1'. Structured types have list of dims, e.g., program variable `vector[10] foo` has entry `('foo', [10])`.

stan_variables() → Dict
Return a dictionary of all Stan program variables. Creates copies of the data in the draws matrix.

stepsize
Stepsize used by sampler for each chain. When sampler algorithm 'fixed_param' is specified, stepsize is None.

summary(*percentiles: List[int] = None*) → pandas.core.frame.DataFrame
Run cmdstan/bin/stansummary over all output csv files. Echo stansummary stdout/stderr to console. Assemble csv tempfile contents into pandasDataFrame.

Parameters **percentiles** – Ordered non-empty list of percentiles to report. Must be integers from (1, 99), inclusive.

validate_csv_files() → None
Checks that csv output files for all chains are consistent. Populates attributes for draws, column_names, num_params, metric_type. Raises exception when inconsistencies detected.

warmup
Deprecated - use "draws(inc_warmup=True)"

8.1.3 CmdStanMLE

class cmdstanpy.**CmdStanMLE**(*runset: cmdstanpy.stanfit.RunSet*)

Container for outputs from CmdStan optimization.

column_names

Names of estimated quantities, includes joint log probability, and all parameters, transformed parameters, and generated quantitites.

optimized_params_dict

Returns optimized params as Dict.

optimized_params_np

Returns optimized params as numpy array.

optimized_params_pd

Returns optimized params as pandas DataFrame.

save_csvfiles(*dir: str = None*) → None

Move output csvfiles to specified directory. If files were written to the temporary session directory, clean filename. E.g., save 'bernoulli-201912081451-1-5nm6as7u.csv' as 'bernoulli-201912081451-1.csv'.

Parameters **dir** – directory path

8.1.4 CmdStanGQ

class cmdstanpy.**CmdStanGQ**(*runset: cmdstanpy.stanfit.RunSet*, *mcmc_sample: pandas.core.frame.DataFrame*)

Container for outputs from CmdStan generate_quantities run.

chains

Number of chains.

column_names

Names of generated quantities of interest.

generated_quantities

A 2-D numpy ndarray which contains generated quantities draws for all chains where the columns correspond to the generated quantities block variables and the rows correspond to the draws from all chains, where first M draws are the first M draws of chain 1 and the last M draws are the last M draws of chain N, i.e., flattened chain, draw ordering.

generated_quantities_pd

Returns the generated quantities as a pandas DataFrame consisting of one column per quantity of interest and one row per draw.

sample_plus_quantities

Returns the column-wise concatenation of the input drawset with generated quantities drawset. If there are duplicate columns in both the input and the generated quantities, the input column is dropped in favor of the recomputed values in the generate quantities drawset.

save_csvfiles(*dir: str = None*) → None

Move output csvfiles to specified directory. If files were written to the temporary session directory, clean filename. E.g., save 'bernoulli-201912081451-1-5nm6as7u.csv' as 'bernoulli-201912081451-1.csv'.

Parameters **dir** – directory path

8.1.5 CmdStanVB

class cmdstanpy.**CmdStanVB**(*runset: cmdstanpy.stanfit.RunSet*)

Container for outputs from CmdStan variational run.

column_names

Names of information items returned by sampler for each draw. Includes approximation information and names of model parameters and computed quantities.

columns

Total number of information items returned by sampler. Includes approximation information and names of model parameters and computed quantities.

save_csvfiles(*dir: str = None*) → None

Move output csvfiles to specified directory. If files were written to the temporary session directory, clean filename. E.g., save 'bernoulli-201912081451-1-5nm6as7u.csv' as 'bernoulli-201912081451-1.csv'.

Parameters **dir** – directory path

variational_params_dict

Returns inferred parameter means as Dict.

variational_params_np

Returns inferred parameter means as numpy array.

variational_params_pd

Returns inferred parameter means as pandas DataFrame.

variational_sample

Returns the set of approximate posterior output draws.

8.1.6 RunSet

class cmdstanpy.stanfit.**RunSet**(*args: cmdstanpy.cmdstan_args.CmdStanArgs*, *chains: int = 4*, *chain_ids: List[int] = None*, *logger: logging.Logger = None*)

Record of CmdStan run for a specified configuration and number of chains.

chain_ids

Chain ids.

chains

Number of chains.

cmds

Per-chain call to CmdStan.

csv_files

List of paths to CmdStan output files.

diagnostic_files

List of paths to CmdStan diagnostic output files.

get_err_msgs() → List[str]

Checks console messages for each chain.

method

Returns the CmdStan method used to generate this fit.

model

Stan model name.

save_csvfiles(*dir: str = None*) → None
Moves csvfiles to specified directory.

Parameters **dir** – directory path

stderr_files
List of paths to CmdStan stderr transcripts.

stdout_files
List of paths to CmdStan stdout transcripts.

genindex

Python Module Index

C

Index

V

W