CmdStanPy Documentation

Release 1.0.0rc2

Stan Development Team

Nov 17, 2021

CONTENTS

CmdStanPy is a lightweight interface to Stan for Python users which provides the necessary objects and functions to do Bayesian inference given a probability model and data. It wraps the CmdStan command line interface in a small set of Python classes which provide methods to do analysis and manage the resulting set of model, data, and posterior estimates.

CHAPTER

ONE

OVERVIEW

CmdStanPy is a lightweight interface to Stan for Python users which provides the necessary objects and functions to do Bayesian inference given a probability model and data. It wraps the CmdStan command line interface in a small set of Python classes which provide methods to do analysis and manage the resulting set of model, data, and posterior estimates. It is lightweight in that it uses minimal memory beyond the memory used by CmdStan. CmdStanPy runs CmdStan, but only instantiates the resulting inference objects in memory upon request. Thus CmdStanPy has the potential to fit more complex models to larger datasets than might be possible in PyStan or RStan.

CmdStan is a file-based interface. CmdStanPy manages the Stan program files and the CmdStan output files. The user can specify the output directory for the CmdStan outputs, otherwise the files will be written to a temporary filesystem which persists throughout the session. This allows the user to test and develop models prospectively, following the Bayesian workflow.

CHAPTER

TWO

INSTALLATION

CmdStanPy is a pure-Python3 package, but it relies on CmdStan for all of its functionality. There are several ways to install CmdStan and CmdStanPy, which depend on the kind of user you are.

2.1 Conda users (Recommended)

If you use conda, installation of both can be done very simply. CmdStanPy and CmdStan are both available via the conda-forge repository.

We recommend creating a new environment for CmdStan[Py]:

```
conda create -n cmdstan -c conda-forge cmdstanpy
```

but installation is possible in an existing environment (Note: you must re-activate the environment after running this command!):

```
conda install -c conda-forge cmdstanpy
```

These commands will install CmdStanPy, CmdStan, and the required compilers for using CmdStan on your system inside a conda environment. To use them, run `conda activate cmdstan`, or whichever name you used for your environment (following `-n` above).

Note that CmdStan is only available on conda for versions 2.26.1 and newer. If you require an older version, you must use one of the following methods to install it. If you require a version of CmdStan *newer* than 2.26.1", but not the latest, you can install it in the standard conda way by specifying `cmdstan==VERSION` in the install command.

2.2 Pip (non-Conda) users

CmdStan can also be installed from PyPI via URL: https://pypi.org/project/cmdstanpy/ or from the command line using `pip`:

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

The optional packages are

- `xarray`, an n-dimension labeled dataset package which can be used for outputs

To install CmdStanPy with all the optional packages:

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

To install the current develop branch from GitHub:

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

If you install CmdStanPy from GitHub, **you must install CmdStan**. The recommended way for Pip users to do so is via the `install_cmdstan` function *described below*

Note: **Note for PyStan & RTools users:** PyStan and CmdStanPy should be installed in separate environments if you are using the RTools toolchain (primarily Windows users). If you already have PyStan installed, you should take care to install CmdStanPy in its own virtual environment.

Jupyter notebook users: CmdStanPy can display progress bars during sampling as well as during the CmdStan build process; these are implemented using the tqdm package, which uses the ipywidgets package in order to update the browser display. For further help on installation and configuration, see ipywidgets installation instructions and this tqdm GitHub issue.

2.3 Installing CmdStan

2.3.1 Prerequisites

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

- Windows: CmdStanPy provides the function `install_cxx_toolchain`
- Linux: install g++ 4.9.3 or clang 6.0. (GNU-Make is the default `make` utility)
- maxOS: install XCode and Xcode command line tools via command: *xcode-select –install*.

2.3.2 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.

The default install location is a hidden directory in the user `$HOME` directory named `.cmdstan`. This directory will be created by the install script.

- From Python

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

- From the command line on Linux or MacOSX

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

- On Windows

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

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

```
install_cmdstan -d my_local_cmdstan -v 2.27.0
ls -F my_local_cmdstan
```

2.3.3 DIY Installation

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

2.3.4 Post Installation: Setting Environment Variables

The default for the CmdStan installation location is a directory named `.cmdstan` in your `$HOME` directory.[1] 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. *Note:* This is done for you if you installed via `conda`, as cmdstan will be installed in the `bin/` subfolder of the environment directory.

```
export CMDSTAN='/path/to/cmdstan-2.27.0'
```

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()
```

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] In very early versions, this hidden directory was named `.cmdstanpy`

CHAPTER

THREE

"HELLO, WORLD"

3.1 Fitting a Stan model using the NUTS-HMC sampler

In order to verify the installation and also to demonstrate the CmdStanPy workflow, we use CmdStanPy to fit the the example Stan model `bernoulli.stan` to the dataset `bernoulli.data.json`. This model and data are included with the CmdStan distribution in subdirectory `examples/bernoulli`. This example allows the user to verify that CmdStanPy, CmdStan, the StanC compiler, and the C++ toolchain have all been properly installed. For substantive example models and guidance on coding statistical models in Stan, see the CmdStan User's Guide.

3.1.1 The Stan model

The model `bernoulli.stan` 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 *CmdStanModel* class manages the Stan program and its corresponding compiled executable. It provides properties and functions to inspect the model code and filepaths. CmdStanPy, manages the environment variable `CMDSTAN` which specifies the path to the local CmdStan installation. The function *cmdstan_path()* returns the value of this environment variable.

```
# import packages
In [1]: import os

In [2]: from cmdstanpy import cmdstan_path, CmdStanModel

# specify Stan program file
In [3]: stan_file = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.stan
↪')
```

(continues on next page)

(continued from previous page)

```
# instantiate the model; compiles the Stan program as needed.
In [4]: model = CmdStanModel(stan_file=stan_file)
INFO:cmdstanpy:found newer exe file, not recompiling

# inspect model object
In [5]: print(model)
CmdStanModel: name=bernoulli
         stan_file=/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/conda/v1.0.
↪0rc2/bin/cmdstan/examples/bernoulli/bernoulli.stan
         exe_file=/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/conda/v1.0.
↪0rc2/bin/cmdstan/examples/bernoulli/bernoulli
         compiler_options=stanc_options={}, cpp_options={}

# inspect compiled model
In [6]: print(model.exe_info())
{'stan_version_major': '2', 'stan_version_minor': '28', 'stan_version_patch': '1', 'STAN_
↪THREADS': 'false', 'STAN_MPI': 'false', 'STAN_OPENCL': 'false', 'STAN_NO_RANGE_CHECKS':
↪ 'false', 'STAN_CPP_OPTIMS': 'false'}
```

3.1.2 Data inputs

CmdStanPy accepts input data either as a Python dictionary which maps data variable names to values, or as the corresponding JSON file.

The bernoulli model requires two inputs: the number of observations N, and an N-length vector y of binary outcomes. The data file *bernoulli.data.json* contains the following inputs:

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

3.1.3 Fitting the model

The *sample()* method 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. The data can be specified either as a filepath or a Python dictionary; in this example, we use the example datafile *bernoulli.data.json*:

By default, the *sample()* method runs 4 sampler chains. The `output_dir` argument is an optional argument which specifies the path to the output directory used by CmdStan. If this argument is omitted, the output files are written to a temporary directory which is deleted when the current Python session is terminated.

```
# specify data file
In [7]: data_file = os.path.join(cmdstan_path(), 'examples', 'bernoulli', 'bernoulli.
↪data.json')

# fit the model
In [8]: fit = model.sample(data=data_file)
INFO:cmdstanpy:CmdStan start procesing
                                                                                    ␣
↪                                                                                   (continues on next page)
↪                                                                                   ␣
↪
```

(continued from previous page)

```
INFO:cmdstanpy:CmdStan done processing.

# printing the object reports sampler commands, output files
In [9]: print(fit)
CmdStanMCMC: model=bernoulli chains=4['method=sample', 'algorithm=hmc', 'adapt',
↪'engaged=1']
 csv_files:
        /tmp/tmprl9js6rj/bernoulli-20211117211908_1.csv
        /tmp/tmprl9js6rj/bernoulli-20211117211908_2.csv
        /tmp/tmprl9js6rj/bernoulli-20211117211908_3.csv
        /tmp/tmprl9js6rj/bernoulli-20211117211908_4.csv
 output_files:
        /tmp/tmprl9js6rj/bernoulli-20211117211908_0-stdout.txt
        /tmp/tmprl9js6rj/bernoulli-20211117211908_1-stdout.txt
        /tmp/tmprl9js6rj/bernoulli-20211117211908_2-stdout.txt
        /tmp/tmprl9js6rj/bernoulli-20211117211908_3-stdout.txt
```

3.1.4 Accessing the sample

The *sample()* method outputs are a set of per-chain Stan CSV files. The filenames follow the template '<model_name>-<YYYYMMDDHHMM>-<chain_id>' plus the file suffix '.csv'. The *CmdStanMCMC* class provides methods to assemble the contents of these files in memory as well as methods to manage the disk files.

Underlyingly, the draws from all chains are stored as an a numpy.ndarray with dimensions: draws, chains, columns. CmdStanPy provides accessor methods which return the sample either in terms of the CSV file columns or in terms of the sampler and Stan program variables. The *draws()* and *draws_pd()* methods return the sample contents in columnar format.

The *stan_variable()* method to returns a numpy.ndarray object which contains the set of all draws in the sample for the named Stan program variable. The draws from all chains are flattened into a single drawset. The first ndarray dimension is the number of draws X number of chains. The remaining ndarray dimensions correspond to the Stan program variable dimension. The *stan_variables()* method returns a Python dict over all Stan model variables.

```
In [10]: fit.draws().shape
Out[10]: (1000, 4, 8)

In [11]: fit.draws(concat_chains=True).shape
Out[11]: (4000, 8)

In [12]: draws_theta = fit.stan_variable(var='theta')

In [13]: draws_theta.shape
Out[13]: (4000,)
```

3.1.5 CmdStan utilities: *stansummary*, *diagnose*

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:

```
In [14]: fit.summary()
Out[14]:
       Mean     MCSE  StdDev     5%   50%   95%   N_Eff  N_Eff/s  R_hat
name
lp__  -7.30  0.0200    0.74 -8.700 -7.00 -6.70  1300.0  13000.0    1.0
theta  0.25  0.0029    0.12  0.079  0.23  0.46  1700.0  16000.0    1.0
```

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.

```
In [15]: print(fit.diagnose())
Processing csv files: /tmp/tmprl9js6rj/bernoulli-20211117211908_1.csv, /tmp/tmprl9js6rj/
↪bernoulli-20211117211908_2.csv, /tmp/tmprl9js6rj/bernoulli-20211117211908_3.csv, /tmp/
↪tmprl9js6rj/bernoulli-20211117211908_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.

Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory.

Effective sample size satisfactory.

Split R-hat values satisfactory all parameters.

Processing complete, no problems detected.
```

3.1.6 Managing Stan CSV files

The *CmdStanMCMC* object keeps track of all output files produced by the sampler run. The *save_csvfiles()* function moves the CSV files to a specified directory.

```
In [16]: fit.save_csvfiles(dir='some/path')
```

3.1.7 Parallelization

The Stan language reduce_sum function provides within-chain parallelization. For models which require computing the sum of a number of independent function evaluations, e.g., when evaluating a number of conditionally independent terms in a log-likelihood, the `reduce_sum` function is used to parallelize this computation.

As of version CmdStan 2.28, it is possible to run the NUTS-HMC sampler on multiple chains from within a single executable using threads. This has the potential to speed up sampling. It also reduces the overall memory footprint required for sampling as all chains share the same copy of data.the input data. When using within-chain parallelization all chains started within a single executable can share all the available threads and once a chain finishes the threads will be reused.

Both within-chain and cross-chain parallelization use the Intel Threading Building Blocks (TBB) library. In order to do either, the Stan model must be compiled with C++ compiler flag `STAN_THREADS`. While any value can be used, we recommend the value `TRUE`.

3.1.8 Progress bar

By default, CmdStanPy displays a progress bar during sampling.

```
In [17]: fit = model.sample(data=data_file)
```

To suppress the progress bar, specify argument `show_progress=False`.

```
In [18]: fit = model.sample(data=data_file, show_progress=False)
```

To see the CmdStan console outputs instead of progress bars, specify `show_console=True`.

```
In [19]: fit = model.sample(data=data_file, show_console=True)
```

This will stream all sampler messages to the console. It provides an alternative way of monitoring progress. In conjunction with Stan programs which contain *print* statments, this provides a way to inspect and debug model behavoir.

3.1.9 Jupyter Lab Notebook requirements

In a Jupyter notebook, this package requires the ipywidgets package. For help on installation and configuration, see ipywidgets installation instructions and this tqdm GitHub issue.

CHAPTER

FOUR

CMDSTANPY WORKFLOW

The statistical modeling enterprise has two principal modalities: development and production. The focus of development is model building, comparison, and validation. Many models are written and fitted to many kinds of data. The focus of production is using a trusted model on real-world data to obtain estimates for decision-making. In both modalities, the essential workflow remains the same: compile a Stan model, assemble input data, do inference on the model conditioned on the data, and validate, access, and export the results.

Model development and testing is an open-ended process, usually requiring many iterations of developing a model, fitting the data, and evaluating the results. Since more user time is spent in model development, CmdStanPy defaults favor development mode. CmdStan is file-based interface. On the assumption that model development will require many successive runs of a model, by default, outputs are written to a temporary directory to avoid filling up the filesystem with unneeded CmdStan output files. Non-default options allow all filepaths to be fully specified so that scripts can be used to distribute analysis jobs across nodes and machines.

The Bayesian workflow for model comparison and model expansion provides a framework for model development, much of which also applies to monitoring model performance in production. The following sections describe the process of building, running, and managing the resulting inference for a single model and set of inputs.

4.1 Compile the Stan model

The: *CmdStanModel* class provides methods to compile and run the Stan program. A CmdStanModel object can be instantiated by specifying either a Stan file or the executable file, or both. If only the Stan file path is specified, the constructor will check for the existence of a correspondingly named exe file in the same directory. If found, it will use this as the exe file path.

By default, when a CmdStanModel object is instantiated from a Stan file, the constructor will compile the model as needed. The constructor argument *compile* controls this behavior.

- `compile=False`: never compile the Stan file.
- `compile=Force`: always compile the Stan file.
- `compile=True`: (default) compile the Stan file as needed, i.e., if no exe file exists or if the Stan file is newer than the exe file.

```
import os
from cmdstanpy import CmdStanModel

my_stanfile = os.path.join('.', 'my_model.stan')
my_model = CmdStanModel(stan_file=my_stanfile)
my_model.name
my_model.stan_file
```

(continues on next page)

(continued from previous page)

```
my_model.exe_file
my_model.code()
```

The CmdStanModel class also provides the *compile()* method, which can be called at any point to (re)compile the model as needed.

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.

In order parallelize within-chain computations using the Stan language `reduce_sum` function, or to parallelize running the NUTS-HMC sampler across chains, the Stan model must be compiled with C++ compiler flag **STAN_THREADS**. While any value can be used, we recommend the value `True`, e.g.:

```
import os
from cmdstanpy import CmdStanModel

my_stanfile = os.path.join('.', 'my_model.stan')
my_model = CmdStanModel(stan_file=my_stanfile, cpp_options={'STAN_THREADS':'true'})
```

4.2 Assemble input and initialization data

CmdStan is file-based interface, therefore all model input and initialization data must be supplied as JSON files, as described in the CmdStan User's Guide.

CmdStanPy inference methods allow inputs and initializations to be specified as in-memory Python dictionary objects which are then converted to JSON via the utility function *cmdstanpy.write_stan_json()*. This method should be used to create JSON input files whenever these inputs contain either a collection compatible with numpy arrays or pandas.Series.

4.3 Run the CmdStan inference engine

For each CmdStan inference method, there is a corresponding method on the *CmdStanModel* class. An example of each is provided in the *next section*

- The *sample()* method runs Stan's HMC-NUTS sampler.

  It returns a *CmdStanMCMC* object which contains a sample from the posterior distribution of the model conditioned on the data.

- The *variational()* method runs Stan's Automatic Differentiation Variational Inference (ADVI) algorithm.

  It returns a *CmdStanVB* object which contains an approximation the posterior distribution in the unconstrained variable space.

- The *optimize()* runs one of Stan's optimization algorithms to find a mode of the density specified by the Stan program.

  It returns a *CmdStanMLE* object.

- The *generate_quantities()* method runs Stan's generate_quantities method which generates additional quantities of interest from a mode. Its take an existing sample as input and uses the parameter estimates in the sample to run the Stan program's generated quantities block.

  It returns a *CmdStanGQ* object.

4.4 Validate, view, export the inference engine outputs

The inference engine results objects *CmdStanMCMC*, *CmdStanVB*, *CmdStanMLE* and `CmdStanGQ`, contain the CmdStan method configuration information and the location of all output files produced. The provide a common set methods for accessing the inference results and metadata, as well as method-specific informational properties and methods.objects

4.4.1 Metadata

By *metadata* we mean the information parsed from the header comments and header row of the Stan CSV files into a *InferenceMetadata* object which is exposed via the object's *metadata* property.

- The metadata *cmdstan_config* property provides the CmdStan configuration information parsed out of the Stan CSV file header.
- The metadata *method_vars_cols* property returns the names, column indices of the inference engine method variables, e.g., the NUTS-HMC sampler output variables are `lp__`, ..., `energy__`.
- The metadata *stan_vars_cols* property returns the names, column indices of all Stan model variables. Container variables will span as many columns, one column per element.
- The metadata *stan_vars_dims* property specifies the names, dimensions of the Stan model variables.

4.4.2 Output data

The CSV data is assembled into the inference result object. CmdStanPy provides accessor methods which return this information either as columnar data (i.e., in terms of the CSV file columns), or as method and model variables.

The *draws()* and *draws_pd()* methods for both *CmdStanMCMC* and *CmdStanGQ* return the sample contents in columnar format, as a numpy.ndarray or pandas.DataFrame, respectively. Similarly, the *draws_xr()* method of these two objects returns the sample contents as an `xarray.Dataset` which maps the method and model variable names to their respective values.

The *method_variables()* method returns a Python dict over all inference method variables.

All inference objects expose the following methods:

The *stan_variable()* method to returns a numpy.ndarray object which contains the set of all draws in the sample for the named Stan program variable. The draws from all chains are flattened into a single drawset. The first ndarray dimension is the number of draws X number of chains. The remaining ndarray dimensions correspond to the Stan program variable dimension. The *stan_variables()* method returns a Python dict over all Stan model variables.

CHAPTER

FIVE

CMDSTANPY EXAMPLES

5.1 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. It 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.

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.

5.1.1 Prerequisites

CmdStanPy displays progress bars during sampling via use of package tqdm. In order for these to display properly, you must have the ipywidgets package installed, and depending on your version of Jupyter or JupyterLab, you must enable it via command:

[1]:
```
!jupyter nbextension enable --py widgetsnbextension
```

```
Enabling notebook extension jupyter-js-widgets/extension...
      - Validating: OK
```

For more information, see the the installation instructions, also this tqdm GitHub issue.

5.1.2 Fitting a model to data

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

We instantiate a `CmdStanModel` from the Stan program file

[2]:
```
import os
from cmdstanpy import CmdStanModel, cmdstan_path

bernoulli_dir = os.path.join(cmdstan_path(), 'examples', 'bernoulli')
stan_file = os.path.join(bernoulli_dir, 'bernoulli.stan')
data_file = os.path.join(bernoulli_dir, 'bernoulli.data.json')
```

(continues on next page)

(continued from previous page)

```
# instantiate, compile bernoulli model
model = CmdStanModel(stan_file=stan_file)
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

By default, the model is compiled during instantiation. The compiled executable is created in the same directory as the program file. If the directory already contains an executable file with a newer timestamp, the model is not recompiled.

We run the sampler on the data using all default settings: 4 chains, each of which runs 1000 warmup and sampling iterations.

[3]:
```
# run CmdStan's sample method, returns object `CmdStanMCMC`
fit = model.sample(data=data_file)
```

```
INFO:cmdstanpy:CmdStan start procesing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

INFO:cmdstanpy:CmdStan done processing.

```

The `sample` method returns a `CmdStanMCMC` object, which contains: - metadata - draws - HMC tuning parameters `metric`, `step_size`

[4]:
```
print('sampler diagnostic variables:\n{}'.format(fit.metadata.method_vars_cols.keys()))
print('stan model variables:\n{}'.format(fit.metadata.stan_vars_cols.keys()))
```

```
sampler diagnostic variables:
dict_keys(['lp__', 'accept_stat__', 'stepsize__', 'treedepth__', 'n_leapfrog__',
↪'divergent__', 'energy__'])
stan model variables:
dict_keys(['theta'])
```

[5]:
```
fit.summary()
```

[5]:
```
        Mean    MCSE  StdDev     5%   50%   95%   N_Eff  N_Eff/s  R_hat
name
lp__   -7.30  0.0200    0.74 -8.700 -7.00 -6.70  1400.0  16000.0    1.0
theta   0.25  0.0034    0.12  0.078  0.23  0.46  1300.0  14000.0    1.0
```

The sampling data from the fit can be accessed either as a `numpy` array or a pandas `DataFrame`:

[6]:
```
print(fit.draws().shape)
fit.draws_pd().head()
```

```
(1000, 4, 8)
```

[6]:
```
      lp__  accept_stat__  stepsize__  treedepth__  n_leapfrog__  divergent__  \
0 -7.34151       1.000000    0.959217          1.0           1.0          0.0
1 -6.75048       1.000000    0.959217          2.0           3.0          0.0
```

(continues on next page)

(continued from previous page)

```
2 -6.75220       0.999568    0.959217          1.0          1.0          0.0
3 -6.91404       0.964206    0.959217          1.0          3.0          0.0
4 -7.37722       0.932651    0.959217          1.0          3.0          0.0

   energy__     theta
0   7.97532  0.132603
1   7.27392  0.241302
2   6.75251  0.238694
3   6.93900  0.182966
4   7.38333  0.129749
```

Additionally, if `xarray` is installed, this data can be accessed another way:

[7]:
```
fit.draws_xr()
```

[7]:
```
<xarray.Dataset>
Dimensions:  (draw: 1000, chain: 4)
Coordinates:
  * chain    (chain) int64 1 2 3 4
  * draw     (draw) int64 0 1 2 3 4 5 6 7 8 ... 992 993 994 995 996 997 998 999
Data variables:
    theta    (chain, draw) float64 0.1326 0.2413 0.2387 ... 0.0839 0.1553 0.1496
Attributes:
    stan_version:        2.28.1
    model:               bernoulli_model
    num_draws_sampling:  1000
```

The `fit` object records the command, the return code, and the paths to the sampler output csv and console files. The string representation of this object displays the CmdStan commands and the location of the 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`.

[8]:
```
fit
```

[8]:
```
CmdStanMCMC: model=bernoulli chains=4['method=sample', 'algorithm=hmc', 'adapt',
↪'engaged=1']
 csv_files:
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_1.csv
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_2.csv
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_3.csv
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_4.csv
 output_files:
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_0-stdout.txt
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_1-stdout.txt
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_2-stdout.txt
        /tmp/tmp3_4hxy0f/bernoulli-20211117211841_3-stdout.txt
```

The sampler output files are written to a temporary directory which is deleted upon session exit unless the `output_dir` argument is specified. The `save_csvfiles` function moves the CmdStan CSV output files to a specified directory without having to re-run the sampler. The console output files are not saved. These files are treated as ephemeral; if the sample is valid, all relevant information is recorded in the CSV files.

5.1.2.1 Sampler Progress

Your model make take a long time to fit. The `sample` method provides two arguments:

- visual progress bar: `show_progress=True`
- stream CmdStan ouput to the console - `show_console=True`

To illustrate how progress bars work, we will run the bernoulli model. Since the progress bars are only visible while the sampler is running and the bernoulli model takes no time at all to fit, we run this model for 200K iterations, in order to see the progress bars in action.

[9]:
```
fit = model.sample(data=data_file, iter_warmup=100000, iter_sampling=100000, show_
↪progress=True)
```

```
INFO:cmdstanpy:CmdStan start procesing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

INFO:cmdstanpy:CmdStan done processing.
```

The Stan language `print` statement can be use to monitor the Stan program state. In order to see this information as the sampler is running, use the `show_console=True` argument. This will stream all CmdStan messages to the terminal while the sampler is running.

[10]:
```
fit = model.sample(data=data_file, chains=2, parallel_chains=1, show_console=True)
```

```
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
INFO:cmdstanpy:Chain [2] start processing
INFO:cmdstanpy:Chain [2] done processing
Chain [1] method = sample (Default)
Chain [1] sample
Chain [1] num_samples = 1000 (Default)
Chain [1] num_warmup = 1000 (Default)
Chain [1] save_warmup = 0 (Default)
Chain [1] thin = 1 (Default)
Chain [1] adapt
Chain [1] engaged = 1 (Default)
Chain [1] gamma = 0.050000000000000003 (Default)
Chain [1] delta = 0.80000000000000004 (Default)
Chain [1] kappa = 0.75 (Default)
Chain [1] t0 = 10 (Default)
Chain [1] init_buffer = 75 (Default)
Chain [1] term_buffer = 50 (Default)
Chain [1] window = 25 (Default)
```

(continues on next page)

(continued from previous page)

```
Chain [1] algorithm = hmc (Default)
Chain [1] hmc
Chain [1] engine = nuts (Default)
Chain [1] nuts
Chain [1] max_depth = 10 (Default)
Chain [1] metric = diag_e (Default)
Chain [1] metric_file =  (Default)
Chain [1] stepsize = 1 (Default)
Chain [1] stepsize_jitter = 0 (Default)
Chain [1] num_chains = 1 (Default)
Chain [1] id = 1 (Default)
Chain [1] data
Chain [1] file = /home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/conda/v1.0.
↪0rc2/bin/cmdstan/examples/bernoulli/bernoulli.data.json
Chain [1] init = 2 (Default)
Chain [1] random
Chain [1] seed = 64832
Chain [1] output
Chain [1] file = /tmp/tmp3_4hxy0f/bernoulli-20211117211850_1.csv
Chain [1] diagnostic_file =  (Default)
Chain [1] refresh = 100 (Default)
Chain [1] sig_figs = -1 (Default)
Chain [1] profile_file = profile.csv (Default)
Chain [1] num_threads = 1 (Default)
Chain [1]
Chain [1]
Chain [1] Gradient evaluation took 4e-06 seconds
Chain [1] 1000 transitions using 10 leapfrog steps per transition would take 0.04␣
↪seconds.
Chain [1] Adjust your expectations accordingly!
Chain [1]
Chain [1]
Chain [1] Iteration:    1 / 2000 [  0%]  (Warmup)
Chain [1] Iteration:  100 / 2000 [  5%]  (Warmup)
Chain [1] Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain [1] Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain [1] Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain [1] Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain [1] Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain [1] Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain [1] Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain [1] Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain [1] Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain [1] Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain [1] Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain [1] Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain [1] Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain [1] Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain [1] Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain [1] Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain [1] Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain [1] Iteration: 1800 / 2000 [ 90%]  (Sampling)
```

(continues on next page)

(continued from previous page)

```
Chain [1] Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain [1] Iteration: 2000 / 2000 [100%]  (Sampling)
Chain [1]
Chain [1] Elapsed Time: 0.008 seconds (Warm-up)
Chain [1] 0.013 seconds (Sampling)
Chain [1] 0.021 seconds (Total)
Chain [1]
Chain [1]
Chain [2] method = sample (Default)
Chain [2] sample
Chain [2] num_samples = 1000 (Default)
Chain [2] num_warmup = 1000 (Default)
Chain [2] save_warmup = 0 (Default)
Chain [2] thin = 1 (Default)
Chain [2] adapt
Chain [2] engaged = 1 (Default)
Chain [2] gamma = 0.050000000000000003 (Default)
Chain [2] delta = 0.80000000000000004 (Default)
Chain [2] kappa = 0.75 (Default)
Chain [2] t0 = 10 (Default)
Chain [2] init_buffer = 75 (Default)
Chain [2] term_buffer = 50 (Default)
Chain [2] window = 25 (Default)
Chain [2] algorithm = hmc (Default)
Chain [2] hmc
Chain [2] engine = nuts (Default)
Chain [2] nuts
Chain [2] max_depth = 10 (Default)
Chain [2] metric = diag_e (Default)
Chain [2] metric_file =  (Default)
Chain [2] stepsize = 1 (Default)
Chain [2] stepsize_jitter = 0 (Default)
Chain [2] num_chains = 1 (Default)
Chain [2] id = 2
Chain [2] data
Chain [2] file = /home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/conda/v1.0.
↪0rc2/bin/cmdstan/examples/bernoulli/bernoulli.data.json
Chain [2] init = 2 (Default)
Chain [2] random
Chain [2] seed = 64832
Chain [2] output
Chain [2] file = /tmp/tmp3_4hxy0f/bernoulli-20211117211850_2.csv
Chain [2] diagnostic_file =  (Default)
Chain [2] refresh = 100 (Default)
Chain [2] sig_figs = -1 (Default)
Chain [2] profile_file = profile.csv (Default)
Chain [2] num_threads = 1 (Default)
Chain [2]
Chain [2]
Chain [2] Gradient evaluation took 4e-06 seconds
Chain [2] 1000 transitions using 10 leapfrog steps per transition would take 0.04␣
↪seconds.
```

(continues on next page)

(continued from previous page)

```
Chain [2] Adjust your expectations accordingly!
Chain [2]
Chain [2]
Chain [2] Iteration:    1 / 2000 [  0%]  (Warmup)
Chain [2] Iteration:  100 / 2000 [  5%]  (Warmup)
Chain [2] Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain [2] Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain [2] Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain [2] Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain [2] Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain [2] Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain [2] Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain [2] Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain [2] Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain [2] Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain [2] Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain [2] Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain [2] Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain [2] Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain [2] Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain [2] Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain [2] Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain [2] Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain [2] Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain [2] Iteration: 2000 / 2000 [100%]  (Sampling)
Chain [2]
Chain [2] Elapsed Time: 0.007 seconds (Warm-up)
Chain [2] 0.015 seconds (Sampling)
Chain [2] 0.022 seconds (Total)
Chain [2]
Chain [2]
```

5.1.3 Running a data-generating model

In this example we use the CmdStan example model data_filegen.stan to generate a simulated dataset given fixed data values.

[11]:
```
model_datagen = CmdStanModel(stan_file='bernoulli_datagen.stan')
datagen_data = {'N':300, 'theta':0.3}
fit_sim = model_datagen.sample(data=datagen_data, fixed_param=True)
fit_sim.summary()
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
INFO:cmdstanpy:CmdStan start procesing
```

```
chain 1 |          | 00:00 Status
```

```

```

```
INFO:cmdstanpy:CmdStan done processing.
```

```

```

[11]:
```
            Mean  MCSE  StdDev  5%  50%    95%   N_Eff   N_Eff/s  R_hat
name
lp__           0   NaN     0.0   0    0    0.0     NaN       NaN    NaN
theta_rep     91  0.25     8.1  77   91  100.0  1100.0  160000.0    1.0
```

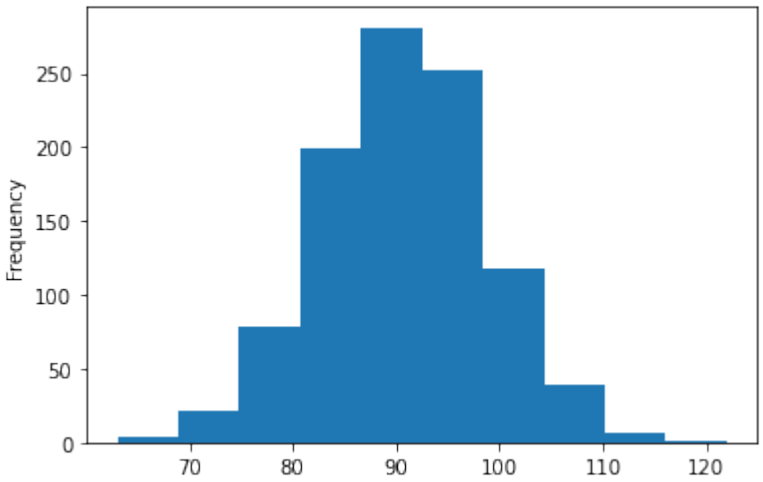
Compute, plot histogram of total successes for `N` Bernoulli trials with chance of success `theta`:

[12]:
```
drawset_pd = fit_sim.draws_pd()
drawset_pd.columns

# restrict to columns over new outcomes of N Bernoulli trials
y_sims = drawset_pd.drop(columns=['lp__', 'accept_stat__'])

# plot total number of successes per draw
y_sums = y_sims.sum(axis=1)
y_sums.astype('int32').plot.hist(range(0,datagen_data['N']+1))
```

[12]:
```
<AxesSubplot:ylabel='Frequency'>
```



5.2 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.

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 estimate 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:

[1]:
```
import os
from cmdstanpy import CmdStanModel, cmdstan_path

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

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

# run CmdStan's otpimize method, returns object `CmdStanMLE`
mle = model.optimize(data=data_file)
print(mle.column_names)
print(mle.optimized_params_dict)
mle.optimized_params_pd
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
```

```
('lp__', 'theta')
OrderedDict([('lp__', -5.00402), ('theta', 0.2)])
```

[1]:
```
      lp__  theta
0 -5.00402    0.2
```

5.3 Variational Inference in Stan

Variational inference is a scalable technique for approximate Bayesian inference. 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.

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

In CmdStanPy, the `CmdStanModel` class method `variational` invokes CmdStan with `method=variational` and returns an estimate of the approximate posterior mean of all model parameters as well as a set of draws from this approximate posterior.

[1]:
```
import os
from cmdstanpy.model import CmdStanModel
from cmdstanpy.utils import cmdstan_path

bernoulli_dir = os.path.join(cmdstan_path(), 'examples', 'bernoulli')
```

(continues on next page)

(continued from previous page)

```
stan_file = os.path.join(bernoulli_dir, 'bernoulli.stan')
data_file = os.path.join(bernoulli_dir, 'bernoulli.data.json')
# instantiate, compile bernoulli model
model = CmdStanModel(stan_file=stan_file)
# run CmdStan's variational inference method, returns object `CmdStanVB`
vi = model.variational(data=data_file)
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
```

The class `CmdStanVB <https://cmdstanpy.readthedocs.io/en/latest/api.html#stanvariational>`__ provides the following properties to access information about the parameter names, estimated means, and the sample: + `column_names` + `variational_params_dict` + `variational_params_np` + `variational_params_pd` + `variational_sample`

[2]:
```
print(vi.column_names)
```

```
('lp__', 'log_p__', 'log_g__', 'theta')
```

[3]:
```
print(vi.variational_params_dict['theta'])
```

```
0.237712
```

[4]:
```
print(vi.variational_sample.shape)
```

```
(1000, 4)
```

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`.

[5]:
```
model_fail = CmdStanModel(stan_file='eta_should_fail.stan')
vi_fail = model_fail.variational()
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/tmp/ipykernel_28130/1235282617.py in <module>
      1 model_fail = CmdStanModel(stan_file='eta_should_fail.stan')
----> 2 vi_fail = model_fail.variational()

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/model.py␣
↪in variational(self, data, seed, inits, output_dir, sig_figs, save_latent_dynamics,␣
↪save_profile, algorithm, iter, grad_samples, elbo_samples, eta, adapt_engaged, adapt_
↪iter, tol_rel_obj, eval_elbo, output_samples, require_converged, show_console, refresh,
↪ time_fmt)
   1388         if len(re.findall(pat, contents)) > 0:
   1389             if require_converged:
-> 1390                 raise RuntimeError(
   1391                     'The algorithm may not have converged.\n'
   1392                     'If you would like to inspect the output, '
```

(continues on next page)

(continued from previous page)

```
RuntimeError: The algorithm may not have converged.
If you would like to inspect the output, re-call with require_converged=False
```

Unless you set `require_converged=False`:

[6]:
```
vi_fail = model_fail.variational(require_converged=False)
```

```
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
WARNING:cmdstanpy:The algorithm may not have converged.
Proceeding because require_converged is set to False
```

This lets you inspect the output to try to diagnose the issue with the model

[7]:
```
vi_fail.variational_params_dict
```

[7]:
```
OrderedDict([('lp__', 0.0),
             ('log_p__', 0.0),
             ('log_g__', 0.0),
             ('mu[1]', 0.0688089),
             ('mu[2]', -0.0598686)])
```

See the api docs, section `CmdStanModel.variational <https://cmdstanpy.readthedocs.io/en/latest/api.html#cmdstanpy.CmdStanModel.variational>`__ for a full description of all arguments.

5.4 Using Variational Estimates to Initialize the NUTS-HMC Sampler

In this example we show how to use the parameter estimates return by Stan's variational inference algorithm as the initial parameter values for Stan's NUTS-HMC sampler. By default, the sampler algorithm randomly initializes all model parameters in the range uniform[-2, 2]. When the true parameter value is outside of this range, starting from the ADVI estimates will speed up and improve adaptation.

5.4.1 Model and data

The Stan model and data are taken from the posteriordb package.

We use the blr model, a Bayesian standard linear regression model with noninformative priors, and its corresponding simulated dataset sblri.json, which was simulated via script sblr.R. For conveince, we have copied the posteriordb model and data to this directory, in files `blr.stan` and `sblri.json`.

[1]:
```
import os
from cmdstanpy import CmdStanModel

stan_file = 'blr.stan' # basic linear regression
data_file = 'sblri.json' # simulated data

model = CmdStanModel(stan_file=stan_file)

print(model.code())
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

```
data {
  int <lower=0> N;
  int <lower=0> D;
  matrix [N, D] X;
  vector [N] y;
}
parameters {
  vector [D] beta;
  real <lower=0> sigma;
}
model {
  // prior
  target += normal_lpdf(beta | 0, 10);
  target += normal_lpdf(sigma | 0, 10);
  // likelihood
  target += normal_lpdf(y | X * beta, sigma);
}
```

5.4.2 Run Stan's variational inference algorithm, obtain fitted estimates

The `CmdStanModel` method `variational` runs CmdStan's ADVI algorithm. Because this algorithm is unstable and may fail to converge, we run it with argument `require_converged` set to `False`. We also specify a seed, to avoid instabilities as well as for reproducibility.

[2]:
```
vb_fit = model.variational(data=data_file, require_converged=False, seed=123)
```

```
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
WARNING:cmdstanpy:The algorithm may not have converged.
Proceeding because require_converged is set to False
```

The ADVI algorithm provides estimates of all model parameters.

The `variational` method returns a `CmdStanVB` object, with method `stan_variables`, which returns the approximate estimates of all model parameters as a Python dictionary.

[3]:
```
print(vb_fit.stan_variables())
```

```
{'beta': array([0.997115, 0.993865, 0.991472, 0.993601, 1.0095  ]), 'sigma': 1.67}
```

Posteriordb provides reference posteriors for all models. For the blr model, conditioned on the dataset `sblri.json`, the reference posteriors are in file sblri-blr.json

The reference posteriors for all elements of `beta` and `sigma` are all very close to 1.0.

The experiments reported in the paper Pathfinder: Parallel quasi-Newton variational inference by Zhang et al. show that mean-field ADVI provides a better estimate of the posterior, as measured by the 1-Wasserstein distance to the reference posterior, than 75 iterations of the warmup Phase I algorithm used by the NUTS-HMC sampler, furthermore, ADVI is more computationally efficient, requiring fewer evaluations of the log density and gradient functions. Therefore, using the estimates from ADVI to initialize the parameter values for the NUTS-HMC sampler will allow the sampler to do a better job of adapting the stepsize and metric during warmup, resulting in better performance and estimation.

[4]:
```
vb_vars = vb_fit.stan_variables()
mcmc_vb_inits_fit = model.sample(
    data=data_file, inits=vb_vars, iter_warmup=75, seed=12345
)
```

```
INFO:cmdstanpy:CmdStan start procesing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

INFO:cmdstanpy:CmdStan done processing.
```

[5]:
```
mcmc_vb_inits_fit.summary()
```

[5]:
```
            Mean      MCSE   StdDev      5%     50%     95%    N_Eff  N_Eff/s  \
name
lp__    -160.00  0.054000  1.80000 -160.00 -160.00 -150.00  1100.00   1000.0
beta[1]    1.00  0.000013  0.00097    1.00    1.00    1.00  5613.00   5255.0
beta[2]    1.00  0.000017  0.00120    1.00    1.00    1.00  4801.00   4496.0
beta[3]    1.00  0.000013  0.00093    1.00    1.00    1.00  5377.00   5035.0
beta[4]    1.00  0.000015  0.00110    1.00    1.00    1.00  4875.00   4565.0
beta[5]    1.00  0.000014  0.00100    1.00    1.00    1.00  5573.00   5219.0
sigma      0.96  0.000000  0.07000    0.86    0.96    1.09   270.85    253.6

          R_hat
name
lp__       1.00
beta[1]    1.00
beta[2]    1.00
beta[3]    1.00
beta[4]    1.00
beta[5]    1.00
sigma      1.01
```

The sampler estimates match the reference posterior.

[6]:
```
print(mcmc_vb_inits_fit.diagnose())
```

```
Processing csv files: /tmp/tmpqs8f3aqr/blr-20211117211903_1.csv, /tmp/tmpqs8f3aqr/blr-
↪20211117211903_2.csv, /tmp/tmpqs8f3aqr/blr-20211117211903_3.csv, /tmp/tmpqs8f3aqr/blr-
↪20211117211903_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.
```

(continues on next page)

(continued from previous page)

```
Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory.

Effective sample size satisfactory.

Split R-hat values satisfactory all parameters.

Processing complete, no problems detected.
```

Using the default random parameter initializations, we need to run more warmup iteratons. If we only run 75 warmup iterations with random inits, the result fails to estimate `sigma` correctly. It is necessary to run the model with at least 150 warmup iterations to produce a good set of estimates.

[7]:
```
mcmc_random_inits_fit = model.sample(data=data_file, iter_warmup=75, seed=12345)
```

```
INFO:cmdstanpy:CmdStan start procesing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

INFO:cmdstanpy:CmdStan done processing.
```

[8]:
```
mcmc_random_inits_fit.summary()
```

[8]:

name	Mean	MCSE	StdDev	5%	50%	95%	N_Eff	N_Eff/s	R_hat
lp__	-190.0	25.00000	36.0000	-230.00	-170.0	-160.0	2.0	4.7	13.0
beta[1]	1.0	0.00012	0.0021	1.00	1.0	1.0	293.0	689.0	1.0
beta[2]	1.0	0.00020	0.0029	0.99	1.0	1.0	204.0	479.0	1.0
beta[3]	1.0	0.00013	0.0021	1.00	1.0	1.0	250.0	587.0	1.0
beta[4]	1.0	0.00013	0.0022	1.00	1.0	1.0	279.0	656.0	1.0
beta[5]	1.0	0.00017	0.0023	1.00	1.0	1.0	180.0	421.0	1.1
sigma	2.0	0.70000	1.1000	0.90	2.7	3.2	2.0	4.8	11.3

[9]:
```
print(mcmc_random_inits_fit.diagnose())
```

```
Processing csv files: /tmp/tmpqs8f3aqr/blr-20211117211904_1.csv, /tmp/tmpqs8f3aqr/blr-
↪20211117211904_2.csv, /tmp/tmpqs8f3aqr/blr-20211117211904_3.csv, /tmp/tmpqs8f3aqr/blr-
↪20211117211904_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
544 of 4000 (14%) transitions ended with a divergence.
These divergent transitions indicate that HMC is not fully able to explore the posterior␣
↪distribution.
```

(continues on next page)

(continued from previous page)

```
Try increasing adapt delta closer to 1.
If this doesn't remove all divergences, try to reparameterize the model.

Checking E-BFMI - sampler transitions HMC potential energy.
The E-BFMI, 0.008, is below the nominal threshold of 0.3 which suggests that HMC may␣
↪have trouble exploring the target distribution.
If possible, try to reparameterize the model.

The following parameters had fewer than 0.001 effective draws per transition:
  sigma
Such low values indicate that the effective sample size estimators may be biased high␣
↪and actual performance may be substantially lower than quoted.

The following parameters had split R-hat greater than 1.1:
  beta[5], sigma
Such high values indicate incomplete mixing and biased estimation.
You should consider regularizating your model with additional prior information or a␣
↪more effective parameterization.

Processing complete.
```

5.5 Generating new quantities of interest.

The generated quantities block computes quantities of interest 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

5.5.1 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 instantiate the model `bernoulli`, as in the "Hello World" section of the CmdStanPy tutorial notebook.

[1]:
```
import os
from cmdstanpy import cmdstan_path, CmdStanModel, CmdStanMCMC, CmdStanGQ

bernoulli_dir = os.path.join(cmdstan_path(), 'examples', 'bernoulli')
stan_file = os.path.join(bernoulli_dir, 'bernoulli.stan')
```

(continues on next page)

(continued from previous page)

```
data_file = os.path.join(bernoulli_dir, 'bernoulli.data.json')

# instantiate, compile bernoulli model
model = CmdStanModel(stan_file=stan_file)
print(model.code())
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

```
data {
  int<lower=0> N;
  array[N] int<lower=0,upper=1> y; // or 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 input data consists of `N` - the number of bernoulli trials and `y` - the list of observed outcomes. Inspection of the data shows that on average, there is a 20% chance of success for any given Bernoulli trial.

[2]:
```
# examine bernoulli data
import ujson
import statistics
with open(data_file,'r') as fp:
    data_dict = ujson.load(fp)
print(data_dict)
print('mean of y: {}'.format(statistics.mean(data_dict['y'])))
```

```
{'N': 10, 'y': [0, 1, 0, 0, 0, 0, 0, 0, 0, 1]}
mean of y: 0.2
```

As in the "Hello World" tutorial, we produce a sample from the posterior of the model conditioned on the data:

[3]:
```
# fit the model to the data
fit = model.sample(data=data_file)
```

```
INFO:cmdstanpy:CmdStan start procesing
```

```
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

INFO:cmdstanpy:CmdStan done processing.
```

The fitted model produces an estimate of `theta` - the chance of success

[4]:
```
fit.summary()
```

[4]:
```
        Mean   MCSE  StdDev     5%   50%   95%   N_Eff  N_Eff/s  R_hat
name
lp__   -7.30  0.020    0.76 -8.800 -7.00 -6.80  1500.0  16000.0    1.0
theta   0.25  0.003    0.12  0.076  0.23  0.47  1500.0  17000.0    1.0
```

To run a prior predictive check, we add a `generated quantities` block to the model, in which we generate a new data vector `y_rep` using the current estimate of theta. The resulting model is in file bernoulli_ppc.stan

[5]:
```
model_ppc = CmdStanModel(stan_file='bernoulli_ppc.stan')
print(model_ppc.code())
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);
  y ~ bernoulli(theta);
}
generated quantities {
  int y_rep[N];
  for (n in 1:N)
    y_rep[n] = bernoulli_rng(theta);
}
```

We run the `generate_quantities` method on `bernoulli_ppc` using existing sample `fit` as input. The `generate_quantities` method takes the values of `theta` in the `fit` sample as the set of draws from the posterior used to generate the corresponsing `y_rep` quantities of interest.

The arguments to the `generate_quantities` method are: + `data` - the data used to fit the model + `mcmc_sample` - either a `CmdStanMCMC` object or a list of stan-csv files

[6]:
```
new_quantities = model_ppc.generate_quantities(data=data_file, mcmc_sample=fit)
```

```
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
INFO:cmdstanpy:Chain [2] start processing
INFO:cmdstanpy:Chain [2] done processing
INFO:cmdstanpy:Chain [3] start processing
INFO:cmdstanpy:Chain [3] done processing
INFO:cmdstanpy:Chain [4] start processing
INFO:cmdstanpy:Chain [4] done processing
```

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.

In this example, each draw consists of the N-length array of replicate of the `bernoulli` model's input variable `y`, which is an N-length array of Bernoulli outcomes.

[7]:
```
print(new_quantities.draws().shape, new_quantities.column_names)
for i in range(3):
    print (new_quantities.draws()[i,:])
```

```
(1000, 4, 10) ('y_rep[1]', 'y_rep[2]', 'y_rep[3]', 'y_rep[4]', 'y_rep[5]', 'y_rep[6]',
↪'y_rep[7]', 'y_rep[8]', 'y_rep[9]', 'y_rep[10]')
[[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 1. 1. 1. 0. 1. 1.]
 [0. 1. 0. 0. 1. 0. 0. 0. 1. 0.]
 [0. 1. 0. 0. 1. 1. 1. 0. 1. 1.]]
[[0. 0. 0. 0. 0. 0. 1. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 1. 0. 0.]]
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 1. 0. 0. 0. 0. 0. 1. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

We can also use `draws_pd(inc_sample=True)` to get a pandas DataFrame which combines the input drawset with the generated quantities.

[8]:
```
sample_plus = new_quantities.draws_pd(inc_sample=True)
print(type(sample_plus),sample_plus.shape)
names = list(sample_plus.columns.values[7:18])
sample_plus.iloc[0:3, :]
```

```
<class 'pandas.core.frame.DataFrame'> (4000, 18)
```

[8]:

	0	1	2	3	4	5	6	7	8	9	\
0	-8.47779	0.837075	1.00389	1.0	3.0	0.0	8.68476	0.074824	0.0	0.0	
1	-6.74819	1.000000	1.00389	2.0	3.0	0.0	8.14939	0.247684	0.0	0.0	
2	-7.12941	0.945643	1.00389	2.0	3.0	0.0	7.15677	0.152670	1.0	0.0	

	10	11	12	13	14	15	16	17
0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

For models as simple as the bernoulli models here, it would be trivial to re-run the sampler and generate a new sample which contains both the estimate of the parameters `theta` as well as `y_rep` values. For models which are difficult to fit, i.e., when producing a sample is computationally expensive, the `generate_quantities` method is preferred.

5.6 Advanced Topic: Using External C++ Functions

This is based on the relevant portion of the CmdStan documentation here

Consider the following Stan model, based on the bernoulli example.

[2]:
```
from cmdstanpy import CmdStanModel
model_external = CmdStanModel(stan_file='bernoulli_external.stan', compile=False)
print(model_external.code())
```

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

As you can see, it features a function declaration for `make_odds`, but no definition. If we try to compile this, we will get an error.

[3]:
```
model_external.compile()
```

```
INFO:cmdstanpy:compiling stan file /home/docs/checkouts/readthedocs.org/user_builds/
↪cmdstanpy/checkouts/v1.0.0rc2/docsrc/examples/bernoulli_external.stan to exe file /
↪home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/docsrc/
↪examples/bernoulli_external
ERROR:cmdstanpy:Stan program failed to compile:
WARNING:cmdstanpy:
--- Translating Stan model to C++ code ---
bin/stanc  --o=/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.
↪0rc2/docsrc/examples/bernoulli_external.hpp /home/docs/checkouts/readthedocs.org/user_
↪builds/cmdstanpy/checkouts/v1.0.0rc2/docsrc/examples/bernoulli_external.stan
Semantic error in '/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/
↪v1.0.0rc2/docsrc/examples/bernoulli_external.stan', line 2, column 2 to column 29:
   -------------------------------------------------
     1:  functions {
     2:    real make_odds(real theta);
           ^
     3:  }
     4:  data {
   -------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Some function is declared without specifying a definition.
make: *** [make/program:50: /home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/
↪checkouts/v1.0.0rc2/docsrc/examples/bernoulli_external.hpp] Error 1

Command ['make', '/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/
↪v1.0.0rc2/docsrc/examples/bernoulli_external']
        error during processing No such file or directory
```

Even enabling the `--allow_undefined` flag to stanc3 will not allow this model to be compiled quite yet.

[4]:
```
model_external.compile(stanc_options={'allow_undefined':True})
```

```
INFO:cmdstanpy:compiling stan file /home/docs/checkouts/readthedocs.org/user_builds/
↪cmdstanpy/checkouts/v1.0.0rc2/docsrc/examples/bernoulli_external.stan to exe file /
↪home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/docsrc/
↪examples/bernoulli_external
```

To resolve this, we need to both tell the Stan compiler an undefined function is okay **and** let C++ know what it should be.

We can provide a definition in a C++ header file by using the `user_header` argument to either the CmdStanModel constructor or the `compile` method.

This will enables the `allow_undefined` flag automatically.

[5]:
```
model_external.compile(user_header='make_odds.hpp')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_27894/728714853.py in <module>
----> 1 model_external.compile(user_header='make_odds.hpp')

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/model.py␣
↪in compile(self, force, stanc_options, cpp_options, user_header, override_options)
    368                 user_header=user_header,
    369             )
--> 370             compiler_options.validate()
    371
    372             if compiler_options != self._compiler_options:

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/compiler_
↪opts.py in validate(self)
    110         self.validate_stanc_opts()
    111         self.validate_cpp_opts()
--> 112         self.validate_user_header()
    113
    114     def validate_stanc_opts(self) -> None:

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/compiler_
↪opts.py in validate_user_header(self)
    184                 and os.path.isfile(self._user_header)
    185             ):
--> 186                 raise ValueError(
    187                     f"User header file {self._user_header} cannot be found"
```

(continues on next page)

(continued from previous page)

```
    188                 )

ValueError: User header file make_odds.hpp cannot be found
```

We can then run this model and inspect the output

[6]:
```
fit = model_external.sample(data={'N':10, 'y':[0,1,0,0,0,0,0,0,0,1]})
fit.stan_variable('odds')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_27894/4268543152.py in <module>
----> 1 fit = model_external.sample(data={'N':10, 'y':[0,1,0,0,0,0,0,0,0,1]})
      2 fit.stan_variable('odds')

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/model.py␣
↪in sample(self, data, chains, parallel_chains, threads_per_chain, seed, chain_ids,␣
↪inits, iter_warmup, iter_sampling, save_warmup, thin, max_treedepth, metric, step_size,
↪ adapt_engaged, adapt_delta, adapt_init_phase, adapt_metric_window, adapt_step_size,␣
↪fixed_param, output_dir, sig_figs, save_latent_dynamics, save_profile, show_progress,␣
↪show_console, refresh, time_fmt, force_one_process_per_chain)
    920         )
    921         with MaybeDictToFilePath(data, inits) as (_data, _inits):
--> 922             args = CmdStanArgs(
    923                 self._name,
    924                 self._exe_file,

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/cmdstan_
↪args.py in __init__(self, model_name, model_exe, chain_ids, method_args, data, seed,␣
↪inits, output_dir, sig_figs, save_latent_dynamics, save_profile, refresh)
    744             self.method = Method.VARIATIONAL
    745         self.method_args.validate(len(chain_ids) if chain_ids else None)
--> 746         self.validate()
    747
    748     def validate(self) -> None:

~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.0rc2/cmdstanpy/cmdstan_
↪args.py in validate(self)
    758             raise ValueError('no stan model specified')
    759         if self.model_exe is None:
--> 760             raise ValueError('model not compiled')
    761
    762         if self.chain_ids is not None:

ValueError: model not compiled
```

The contents of this header file are a bit complicated unless you are familiar with the C++ internals of Stan, so they are presented without comment:

```
#include <boost/math/tools/promotion.hpp>
#include <ostream>

namespace bernoulli_model_namespace {
```

(continues on next page)

(continued from previous page)

```
    template <typename T0__>  inline  typename
          boost::math::tools::promote_args<T0__>::type
          make_odds(const T0__& theta, std::ostream* pstream__) {
            return theta / (1 - theta);
       }
}
```

CHAPTER

SIX

API REFERENCE

The following documents the public API of CmdStanPy. It is expected to be stable between versions, with backwards compatibility between minor versions and deprecation warnings preceeding breaking changes. There is also the *internal API*, which is makes no such guarantees.

6.1 Internal API Reference

The following documents the internal API of CmdStanPy. No guarantees are made about backwards compatibility between minor versions and refactors are expected. If you find yourself needing something exposed here, please open an issue requesting it be added to the *public API*.

6.1.1 Classes

6.1.1.1 InferenceMetadata

class cmdstanpy.**InferenceMetadata**(*config*)

CmdStan configuration and contents of output file parsed out of the Stan CSV file header comments and column headers. Assumes valid CSV files.

Parameters **config** (*Dict[str, Any]*) –

Return type None

property cmdstan_config: Dict[str, Any]

Returns a dictionary containing a set of name, value pairs parsed out of the Stan CSV file header. These include the command configuration and the CSV file header row information. Uses deepcopy for immutability.

property method_vars_cols: Dict[str, Tuple[int, ...]]

Returns a map from a Stan inference method variable to a tuple of column indices in inference engine's output array. Method variable names always end in __, e.g. *lp__*. Uses deepcopy for immutability.

property stan_vars_cols: Dict[str, Tuple[int, ...]]

Returns a map from a Stan program variable name to a tuple of the column indices in the vector or matrix of estimates produced by a CmdStan inference method. Uses deepcopy for immutability.

property stan_vars_dims: Dict[str, Tuple[int, ...]]

Returns map from Stan program variable names to variable dimensions. Scalar types are mapped to the empty tuple, e.g., program variable `int foo` has dimension `()` and program variable `vector[10] bar` has single dimension `(10)`. Uses deepcopy for immutability.

6.1.1.2 RunSet

class cmdstanpy.stanfit.**RunSet**(*args*, *chains=1*, ***, *chain_ids=None*, *time_fmt='%Y%m%d%H%M%S'*, *one_process_per_chain=True*)

Encapsulates the configuration and results of a call to any CmdStan inference method. Records the method return code and locations of all console, error, and output files.

RunSet objects are instantiated by the CmdStanModel class inference methods which validate all inputs, therefore "__init__" method skips input checks.

Parameters

- **args** (cmdstanpy.cmdstan_args.CmdStanArgs) –
- **chains** (*int*) –
- **chain_ids** (*Optional[List[int]]*) –
- **time_fmt** (*str*) –
- **one_process_per_chain** (*bool*) –

Return type None

cmd(*idx*)

Assemble CmdStan invocation. When running parallel chains from single process (2.28 and up), specify CmdStan arg *num_chains* and leave chain idx off CSV files.

Parameters **idx** (*int*) –

Return type List[str]

get_err_msgs()

Checks console messages for each CmdStan run.

Return type str

save_csvfiles(*dir=None*)

Moves CSV files to specified directory.

Parameters **dir** (*Optional[str]*) – directory path

Return type None

See also:

cmdstanpy.from_csv

property chain_ids: List[int]

Chain ids.

property chains: int

Number of chains.

property csv_files: List[str]

List of paths to CmdStan output files.

property diagnostic_files: List[str]

List of paths to CmdStan hamiltonian diagnostic files.

property method: cmdstanpy.cmdstan_args.Method

CmdStan method used to generate this fit.

property model: str

Stan model name.

property num_procs: int
Number of processes run.

property one_process_per_chain: bool
When True, for each chain, call CmdStan in its own subprocess. When False, use CmdStan's *num_chains* arg to run parallel chains. Always True if CmdStan < 2.28. For CmdStan 2.28 and up, *sample* method determines value.

property profile_files: List[str]
List of paths to CmdStan profiler files.

property stdout_files: List[str]
List of paths to transcript of CmdStan messages sent to the console. Transcripts include config information, progress, and error messages.

6.1.1.3 CompilerOptions

class cmdstanpy.compiler_opts.**CompilerOptions**(*, *stanc_options=None*, *cpp_options=None*, *user_header=None*)
User-specified flags for stanc and C++ compiler.

Parameters
- **stanc_options** (*Optional[Dict[str, Any]]*) –
- **cpp_options** (*Optional[Dict[str, Any]]*) –
- **user_header** (*Optional[str]*) –

Return type None

stanc_options - stanc compiler flags, options

cpp_options - makefile options

Type NAME=value

user_header - path to a user .hpp file to include during compilation

add(*new_opts*)
Adds options to existing set of compiler options.

Parameters **new_opts** (cmdstanpy.compiler_opts.CompilerOptions) –

Return type None

add_include_path(*path*)
Adds include path to existing set of compiler options.

Parameters **path** (*str*) –

Return type None

compose()
Format makefile options as list of strings.

Return type List[str]

is_empty()
True if no options specified.

Return type bool

validate()
Check compiler args. Raise ValueError if invalid options are found.

Return type None

validate_cpp_opts()
Check cpp compiler args. Raise ValueError if bad config is found.

Return type None

validate_stanc_opts()
Check stanc compiler args and consistency between stanc and C++ options. Raise ValueError if bad config is found.

Return type None

validate_user_header()
User header exists. Raise ValueError if bad config is found.

Return type None

property cpp_options: Dict[str, Union[bool, int]]
C++ compiler options.

property stanc_options: Dict[str, Union[bool, int, str]]
Stanc compiler options.

property user_header: str
user header.

6.1.1.4 CmdStanArgs

class cmdstanpy.cmdstan_args.**CmdStanArgs**(*model_name*, *model_exe*, *chain_ids*, *method_args*, *data=None*, *seed=None*, *inits=None*, *output_dir=None*, *sig_figs=None*, *save_latent_dynamics=False*, *save_profile=False*, *refresh=None*)
Container for CmdStan command line arguments. Consists of arguments common to all methods and and an object which contains the method-specific arguments.

Parameters

- **model_name** (*str*) –
- **model_exe** (*Optional[str]*) –
- **chain_ids** (*Optional[List[int]]*) –
- **method_args** (*Union[cmdstanpy.cmdstan_args.SamplerArgs, cmdstanpy.cmdstan_args.OptimizeArgs, cmdstanpy.cmdstan_args.GenerateQuantitiesArgs, cmdstanpy.cmdstan_args.VariationalArgs]*) –
- **data** (*Optional[Union[Mapping[str, Any], str]]*) –
- **seed** (*Optional[Union[int, List[int]]]*) –
- **inits** (*Optional[Union[int, float, str, List[str]]]*) –
- **output_dir** (*Optional[str]*) –
- **sig_figs** (*Optional[int]*) –
- **save_latent_dynamics** (*bool*) –
- **save_profile** (*bool*) –
- **refresh** (*Optional[int]*) –

Return type None

compose_command(*idx*, *csv_file*, *, *diagnostic_file=None*, *profile_file=None*, *num_chains=None*)
Compose CmdStan command for non-default arguments.

Parameters

- **idx** (*int*) –
- **csv_file** (*str*) –
- **diagnostic_file** (*Optional[str]*) –
- **profile_file** (*Optional[str]*) –
- **num_chains** (*Optional[int]*) –

Return type List[str]

validate()
Check arguments correctness and consistency.

- input files must exist
- output files must be in a writeable directory
- if no seed specified, set random seed.
- length of per-chain lists equals specified # of chains

Return type None

6.1.1.5 SamplerArgs

class cmdstanpy.cmdstan_args.**SamplerArgs**(*iter_warmup=None*, *iter_sampling=None*, *save_warmup=False*, *thin=None*, *max_treedepth=None*, *metric=None*, *step_size=None*, *adapt_engaged=True*, *adapt_delta=None*, *adapt_init_phase=None*, *adapt_metric_window=None*, *adapt_step_size=None*, *fixed_param=False*)
Arguments for the NUTS adaptive sampler.

Parameters

- **iter_warmup** (*Optional[int]*) –
- **iter_sampling** (*Optional[int]*) –
- **save_warmup** (*bool*) –
- **thin** (*Optional[int]*) –
- **max_treedepth** (*Optional[int]*) –
- **metric** (*Optional[Union[str, Dict[str, Any], List[str], List[Dict[str, Any]]]]*) –
- **step_size** (*Optional[Union[float, List[float]]]*) –
- **adapt_engaged** (*bool*) –
- **adapt_delta** (*Optional[float]*) –
- **adapt_init_phase** (*Optional[int]*) –
- **adapt_metric_window** (*Optional[int]*) –
- **adapt_step_size** (*Optional[int]*) –

- **fixed_param** (*bool*) –

Return type None

compose(*idx*, *cmd*)
Compose CmdStan command for method-specific non-default arguments.

Parameters

- **idx** (*int*) –
- **cmd** (*List[str]*) –

Return type List[str]

validate(*chains*)
Check arguments correctness and consistency.

- adaptation and warmup args are consistent
- if file(s) for metric are supplied, check contents.
- length of per-chain lists equals specified # of chains

Parameters **chains** (*Optional[int]*) –

Return type None

6.1.1.6 OptimizeArgs

class cmdstanpy.cmdstan_args.**OptimizeArgs**(*algorithm=None*, *init_alpha=None*, *iter=None*, *save_iterations=False*, *tol_obj=None*, *tol_rel_obj=None*, *tol_grad=None*, *tol_rel_grad=None*, *tol_param=None*, *history_size=None*)
Container for arguments for the optimizer.

Parameters

- **algorithm** (*Optional[str]*) –
- **init_alpha** (*Optional[float]*) –
- **iter** (*Optional[int]*) –
- **save_iterations** (*bool*) –
- **tol_obj** (*Optional[float]*) –
- **tol_rel_obj** (*Optional[float]*) –
- **tol_grad** (*Optional[float]*) –
- **tol_rel_grad** (*Optional[float]*) –
- **tol_param** (*Optional[float]*) –
- **history_size** (*Optional[int]*) –

Return type None

compose(*idx*, *cmd*)
compose command string for CmdStan for non-default arg values.

Parameters

- **idx** (*int*) –

- **cmd** (*List[str]*) –

Return type List[str]

validate(*chains=None*)
Check arguments correctness and consistency.

Parameters **chains** (*Optional[int]*) –

Return type None

6.1.1.7 VariationalArgs

class cmdstanpy.cmdstan_args.**VariationalArgs**(*algorithm=None*, *iter=None*, *grad_samples=None*, *elbo_samples=None*, *eta=None*, *adapt_iter=None*, *adapt_engaged=True*, *tol_rel_obj=None*, *eval_elbo=None*, *output_samples=None*)

Arguments needed for variational method.

Parameters

- **algorithm** (*Optional[str]*) –
- **iter** (*Optional[int]*) –
- **grad_samples** (*Optional[int]*) –
- **elbo_samples** (*Optional[int]*) –
- **eta** (*Optional[float]*) –
- **adapt_iter** (*Optional[int]*) –
- **adapt_engaged** (*bool*) –
- **tol_rel_obj** (*Optional[float]*) –
- **eval_elbo** (*Optional[int]*) –
- **output_samples** (*Optional[int]*) –

Return type None

compose(*idx*, *cmd*)
Compose CmdStan command for method-specific non-default arguments.

Parameters

- **idx** (*int*) –
- **cmd** (*List[str]*) –

Return type List[str]

validate(*chains=None*)
Check arguments correctness and consistency.

Parameters **chains** (*Optional[int]*) –

Return type None

6.2 Classes

6.2.1 CmdStanModel

A CmdStanModel object encapsulates the Stan program. It manages program compilation and provides the following inference methods:

sample() runs the HMC-NUTS sampler to produce a set of draws from the posterior distribution.

optimize() produce a penalized maximum likelihood estimate (point estimate) of the model parameters.

variational() run CmdStan's variational inference algorithm to approximate the posterior distribution.

generate_quantities() runs CmdStan's generate_quantities method to produce additional quantities of interest based on draws from an existing sample.

class cmdstanpy.**CmdStanModel**(*model_name=None*, *stan_file=None*, *exe_file=None*, *compile=True*, *stanc_options=None*, *cpp_options=None*, *user_header=None*)

The constructor method allows model instantiation given either the Stan program source file or the compiled executable, or both. By default, the constructor will compile the Stan program on instantiation unless the argument `compile=False` is specified. The set of constructor arguments are:

Parameters

- **model_name** (*Optional[str]*) – Model name, used for output file names. Optional, default is the base filename of the Stan program file.
- **stan_file** (*Optional[str]*) – Path to Stan program file.
- **exe_file** (*Optional[str]*) – Path to compiled executable file. Optional, unless no Stan program file is specified. If both the program file and the compiled executable file are specified, the base filenames must match, (but different directory locations are allowed).
- **compile** (*Union[bool, str]*) – Whether or not to compile the model. Default is `True`. If set to the string `"force"`, it will always compile even if an existing executable is found.
- **stanc_options** (*Optional[Dict[str, Any]]*) – Options for stanc compiler, specified as a Python dictionary containing Stanc3 compiler option name, value pairs. Optional.
- **cpp_options** (*Optional[Dict[str, Any]]*) – Options for C++ compiler, specified as a Python dictionary containing C++ compiler option name, value pairs. Optional.
- **user_header** (*Optional[str]*) – A path to a header file to include during C++ compilation. Optional.

Return type None

code()

Return Stan program as a string.

Return type Optional[str]

compile(*force=False*, *stanc_options=None*, *cpp_options=None*, *user_header=None*, *override_options=False*)

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` or unless the compiler options have been changed.

Parameters

- **force** (*bool*) – 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.
- **stanc_options** (*Optional[Dict[str, Any]]*) – Options for stanc compiler.
- **cpp_options** (*Optional[Dict[str, Any]]*) – Options for C++ compiler.
- **user_header** (*Optional[str]*) – A path to a header file to include during C++ compilation.
- **override_options** (*bool*) – When `True`, override existing option. When `False`, add/replace existing options. Default is `False`.

Return type None

exe_info()

Run model with option 'info'. Parse output statements, which all have form 'key = value' into a Dict. If exe file compiled with CmdStan < 2.27, option 'info' isn't available and the method returns an empty dictionary.

Return type Dict[str, str]

generate_quantities(*data=None*, *mcmc_sample=None*, *seed=None*, *gq_output_dir=None*, *sig_figs=None*, *show_console=False*, *refresh=None*, *time_fmt='%Y%m%d%H%M%S'*)

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** (*Optional[Union[Mapping[str, Any], str]]*) – 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** (*Optional[Union[cmdstanpy.stanfit.CmdStanMCMC, List[str]]]*) – 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** (*Optional[int]*) – 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** (*Optional[str]*) – 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.

- **sig_figs** (*Optional[int]*) – Numerical precision used for output CSV and text files. Must be an integer between 1 and 18. If unspecified, the default precision for the system file I/O is used; the usual value is 6. Introduced in CmdStan-2.25.
- **show_console** (*bool*) – If `True`, stream CmdStan messages sent to stdout and stderr to the console. Default is `False`.
- **refresh** (*Optional[int]*) – Specify the number of iterations CmdStan will take between progress messages. Default value is 100.
- **time_fmt** (*str*) – A format string passed to `strftime()` to decide the file names for output CSVs. Defaults to "%Y%m%d%H%M%S"

Returns CmdStanGQ object

Return type *cmdstanpy.stanfit.CmdStanGQ*

optimize(*data=None*, *seed=None*, *inits=None*, *output_dir=None*, *sig_figs=None*, *save_profile=False*, *algorithm=None*, *init_alpha=None*, *tol_obj=None*, *tol_rel_obj=None*, *tol_grad=None*, *tol_rel_grad=None*, *tol_param=None*, *history_size=None*, *iter=None*, *save_iterations=False*, *require_converged=True*, *show_console=False*, *refresh=None*, *time_fmt='%Y%m%d%H%M%S'*)

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. 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** (*Optional[Union[Mapping[str, Any], str]]*) – 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** (*Optional[int]*) – 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** (*Optional[Union[Dict[str, float], float, str]]*) – 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** (*Optional[str]*) – 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.
- **sig_figs** (*Optional[int]*) – Numerical precision used for output CSV and text files. Must be an integer between 1 and 18. If unspecified, the default precision for the system file I/O is used; the usual value is 6. Introduced in CmdStan-2.25.
- **save_profile** (*bool*) – Whether or not to profile auto-diff operations in labelled blocks of code. If `True`, CSV outputs are written to file '<model_name>-<YYYYMMDDHHMM>-profile-<chain_id>'. Introduced in CmdStan-2.26.
- **algorithm** (*Optional[str]*) – Algorithm to use. One of: 'BFGS', 'LBFGS', 'Newton'
- **init_alpha** (*Optional[float]*) – Line search step size for first iteration
- **tol_obj** (*Optional[float]*) – Convergence tolerance on changes in objective function value
- **tol_rel_obj** (*Optional[float]*) – Convergence tolerance on relative changes in objective function value
- **tol_grad** (*Optional[float]*) – Convergence tolerance on the norm of the gradient
- **tol_rel_grad** (*Optional[float]*) – Convergence tolerance on the relative norm of the gradient
- **tol_param** (*Optional[float]*) – Convergence tolerance on changes in parameter value
- **history_size** (*Optional[int]*) – Size of the history for LBFGS Hessian approximation. The value should be less than the dimensionality of the parameter space. 5-10 usually sufficient
- **iter** (*Optional[int]*) – Total number of iterations
- **save_iterations** (*bool*) – When `True`, save intermediate approximations to the output CSV file. Default is `False`.
- **require_converged** (*bool*) – Whether or not to raise an error if Stan reports that "The algorithm may not have converged".
- **show_console** (*bool*) – If `True`, stream CmdStan messages sent to stdout and stderr to the console. Default is `False`.
- **refresh** (*Optional[int]*) – Specify the number of iterations cmdstan will take between progress messages. Default value is 100.
- **time_fmt** (*str*) – A format string passed to `strftime()` to decide the file names for output CSVs. Defaults to "%Y%m%d%H%M%S"

Returns CmdStanMLE object

Return type *cmdstanpy.stanfit.CmdStanMLE*

sample(*data=None*, *chains=None*, *parallel_chains=None*, *threads_per_chain=None*, *seed=None*, *chain_ids=None*, *inits=None*, *iter_warmup=None*, *iter_sampling=None*, *save_warmup=False*, *thin=None*, *max_treedepth=None*, *metric=None*, *step_size=None*, *adapt_engaged=True*, *adapt_delta=None*, *adapt_init_phase=None*, *adapt_metric_window=None*, *adapt_step_size=None*, *fixed_param=False*, *output_dir=None*, *sig_figs=None*, *save_latent_dynamics=False*, *save_profile=False*, *show_progress=True*, *show_console=False*, *refresh=None*, *time_fmt='%Y%m%d%H%M%S'*, ***, *force_one_process_per_chain=None*)

Run or more chains of the NUTS-HMC 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** (*Optional[Union[Mapping[str, Any], str]]*) – 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** (*Optional[int]*) – Number of sampler chains, must be a positive integer.
- **parallel_chains** (*Optional[int]*) – Number of processes to run in parallel. Must be a positive integer. Defaults to `multiprocessing.cpu_count()`, i.e., it will only run as many chains in parallel as there are cores on the machine. Note that CmdStan 2.28 and higher can run all chains in parallel providing that the model was compiled with threading support.
- **threads_per_chain** (*Optional[int]*) – 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. For such models, CmdStan version 2.28 and higher will run all chains in parallel from within a single process. The total number of threads used will be `parallel_chains * threads_per_chain`, where the default value for parallel_chains is the number of cpus, not chains.
- **seed** (*Optional[Union[int, List[int]]]*) – 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** (*Optional[Union[int, List[int]]]*) – 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** (*Optional[Union[Dict[str, float], float, str, List[str]]]*) – 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** (*Optional[int]*) – Number of warmup iterations for each chain.
- **iter_sampling** (*Optional[int]*) – Number of draws from the posterior for each chain.
- **save_warmup** (*bool*) – When `True`, sampler saves warmup draws as part of the Stan CSV output file.
- **thin** (*Optional[int]*) – Period between recorded iterations. Default is 1, i.e., all iterations are recorded.
- **max_treedepth** (*Optional[int]*) – Maximum depth of trees evaluated by NUTS sampler per iteration.
- **metric** (*Optional[Union[str, Dict[str, Any], List[str], List[Dict[str, Any]]]]*) – 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.

  If the value of the metric argument is a Python dict object, it must contain an entry 'inv_metric' which specifies either the diagnoal or dense matrix.

  If the value of the metric argument is a list of Python dicts, its length must match the number of chains and all dicts must containan entry 'inv_metric' and all 'inv_metric' entries must have the same shape.

- **step_size** (*Optional[Union[float, List[float]]]*) – Initial step size 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** (*bool*) – When `True`, adapt step size and metric.
- **adapt_delta** (*Optional[float]*) – 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** (*Optional[int]*) – Iterations for initial phase of adaptation during which step size is adjusted so that the chain converges towards the typical set.
- **adapt_metric_window** (*Optional[int]*) – 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** (*Optional[int]*) – Number of iterations given over to adjusting the step size given the tuned metric during the final phase of adaptation.
- **fixed_param** (*bool*) – 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** (*Optional[str]*) – 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.
- **sig_figs** (*Optional[int]*) – Numerical precision used for output CSV and text files. Must be an integer between 1 and 18. If unspecified, the default precision for the system file I/O is used; the usual value is 6. Introduced in CmdStan-2.25.
- **save_latent_dynamics** (*bool*) – Whether or not to output the position and momentum information for the model parameters (unconstrained). If `True`, CSV outputs are written to an output file '<model_name>-<YYYYMMDDHHMM>-diagnostic-<chain_id>', e.g. 'bernoulli-201912081451-diagnostic-1.csv', see https://mc-stan.org/docs/cmdstan-guide/stan-csv.html, section "Diagnostic CSV output file" for details.
- **save_profile** (*bool*) – Whether or not to profile auto-diff operations in labelled blocks of code. If `True`, CSV outputs are written to file '<model_name>-<YYYYMMDDHHMM>-profile-<chain_id>'. Introduced in CmdStan-2.26, see https://mc-stan.org/docs/cmdstan-guide/stan-csv.html, section "Profiling CSV output file" for details.
- **show_progress** (*bool*) – If `True`, display progress bar to track progress for warmup and sampling iterations. Default is `True`, unless package tqdm progress bar encounter errors.
- **show_console** (*bool*) – If `True`, stream CmdStan messages sent to stdout and stderr to the console. Default is `False`.
- **refresh** (*Optional[int]*) – Specify the number of iterations CmdStan will take between progress messages. Default value is 100.
- **time_fmt** (*str*) – A format string passed to `strftime()` to decide the file names for output CSVs. Defaults to "%Y%m%d%H%M%S"
- **force_one_process_per_chain** (*Optional[bool]*) – If `True`, run multiple chains in distinct processes regardless of model ability to run parallel chains (CmdStan 2.28+ feature). If `False`, always run multiple chains in one process (does not check that this is valid).

  If None (Default): Check that CmdStan version is >=2.28, and that model was compiled with STAN_THREADS=True, and utilize the parallel chain functionality if those conditions are met.

Returns CmdStanMCMC object

Return type *cmdstanpy.stanfit.CmdStanMCMC*

src_info()

Run stanc with option '–info'.

If stanc is older than 2.27 or if the stan file cannot be found, returns an empty dictionary.

Return type Dict[str, Any]

variational(*data=None*, *seed=None*, *inits=None*, *output_dir=None*, *sig_figs=None*, *save_latent_dynamics=False*, *save_profile=False*, *algorithm=None*, *iter=None*, *grad_samples=None*, *elbo_samples=None*, *eta=None*, *adapt_engaged=True*, *adapt_iter=None*, *tol_rel_obj=None*, *eval_elbo=None*, *output_samples=None*, *require_converged=True*, *show_console=False*, *refresh=None*, *time_fmt='%Y%m%d%H%M%S'*)

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** (*Optional[Union[Mapping[str, Any], str]]*) – 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** (*Optional[int]*) – 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** (*Optional[float]*) – 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** (*Optional[str]*) – 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.
- **sig_figs** (*Optional[int]*) – Numerical precision used for output CSV and text files. Must be an integer between 1 and 18. If unspecified, the default precision for the system file I/O is used; the usual value is 6. Introduced in CmdStan-2.25.
- **save_latent_dynamics** (*bool*) – Whether or not to save diagnostics. If `True`, CSV outputs are written to output file '<model_name>-<YYYYMMDDHHMM>-diagnostic-<chain_id>', e.g. 'bernoulli-201912081451-diagnostic-1.csv'.
- **save_profile** (*bool*) – Whether or not to profile auto-diff operations in labelled blocks of code. If `True`, CSV outputs are written to file '<model_name>-<YYYYMMDDHHMM>-profile-<chain_id>'. Introduced in CmdStan-2.26.
- **algorithm** (*Optional[str]*) – Algorithm to use. One of: 'meanfield', 'fullrank'.
- **iter** (*Optional[int]*) – Maximum number of ADVI iterations.
- **grad_samples** (*Optional[int]*) – Number of MC draws for computing the gradient. Default is 10. If problems arise, try doubling current value.
- **elbo_samples** (*Optional[int]*) – Number of MC draws for estimate of ELBO.
- **eta** (*Optional[float]*) – Step size scaling parameter.
- **adapt_engaged** (*bool*) – Whether eta adaptation is engaged.
- **adapt_iter** (*Optional[int]*) – Number of iterations for eta adaptation.
- **tol_rel_obj** (*Optional[float]*) – Relative tolerance parameter for convergence.
- **eval_elbo** (*Optional[int]*) – Number of iterations between ELBO evaluations.

- **output_samples** (*Optional[int]*) – Number of approximate posterior output draws to save.
- **require_converged** (*bool*) – Whether or not to raise an error if Stan reports that "The algorithm may not have converged".
- **show_console** (*bool*) – If `True`, stream CmdStan messages sent to stdout and stderr to the console. Default is `False`.
- **refresh** (*Optional[int]*) – Specify the number of iterations CmdStan will take between progress messages. Default value is 100.
- **time_fmt** (*str*) – A format string passed to `strftime()` to decide the file names for output CSVs. Defaults to "%Y%m%d%H%M%S"

Returns CmdStanVB object

Return type *cmdstanpy.stanfit.CmdStanVB*

property cpp_options: Dict[str, Union[bool, int]]
Options to C++ compilers.

property exe_file: Optional[str]
Full path to Stan exe file.

property name: str
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`.

property stan_file: Optional[str]
Full path to Stan program file.

property stanc_options: Dict[str, Union[bool, int, str]]
Options to stanc compilers.

property user_header: str
The user header file if it exists, otherwise empty

6.2.2 CmdStanMCMC

class cmdstanpy.**CmdStanMCMC**(*runset*)
Container for outputs from CmdStan sampler run. Provides methods to summarize and diagnose the model fit and accessor methods to access the entire sample or individual items. Created by *CmdStanModel.sample()*

The sample is lazily instantiated on first access of either the resulting sample or the HMC tuning parameters, i.e., the step size and metric.

Parameters **runset** (cmdstanpy.stanfit.RunSet) –

Return type None

diagnose()
Run cmdstan/bin/diagnose over all output CSV files, return console output.

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

Return type Optional[str]

draws(*, *inc_warmup=False*, *concat_chains=False*)

Returns a numpy.ndarray over all draws from all chains which is stored column major so that the values for a parameter are contiguous in memory, likewise all draws from a chain are contiguous. By default, returns a 3D array arranged (draws, chains, columns); parameter `concat_chains=True` will return a 2D array where all chains are flattened into a single column, preserving chain order, so that given M chains of N draws, the first N draws are from chain 1, up through the last N draws from chain M.

Parameters

- **inc_warmup** (*bool*) – 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`.
- **concat_chains** (*bool*) – When `True` return a 2D array flattening all all draws from all chains. Default value is `False`.

Return type numpy.ndarray

See also:

CmdStanMCMC.draws_pd, *CmdStanMCMC.draws_xr*, *CmdStanGQ.draws*

draws_pd(*vars=None*, *inc_warmup=False*)

Returns the sample draws as a pandas DataFrame. Flattens all chains into single column. Container variables (array, vector, matrix) will span multiple columns, one column per element. E.g. variable 'matrix[2,2] foo' spans 4 columns: 'foo[1,1], ... foo[2,2]'.

Parameters

- **vars** (*Optional[Union[List[str], str]]*) – optional list of variable names.
- **inc_warmup** (*bool*) – 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`.

Return type pandas.core.frame.DataFrame

See also:

CmdStanMCMC.draws, *CmdStanMCMC.draws_xr*, *CmdStanGQ.draws_pd*

draws_xr(*vars=None*, *inc_warmup=False*)

Returns the sampler draws as a xarray Dataset.

Parameters

- **vars** (*Optional[Union[List[str], str]]*) – optional list of variable names.
- **inc_warmup** (*bool*) – 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`.

Return type xarray.core.dataset.Dataset

See also:

CmdStanMCMC.draws, *CmdStanMCMC.draws_pd*, *CmdStanGQ.draws_xr*

`method_variables`()
Returns a dictionary of all sampler variables, i.e., all output column names ending in __. Assumes that all variables are scalar variables where column name is variable name. Maps each column name to a numpy.ndarray (draws x chains x 1) containing per-draw diagnostic values.

Return type Dict[str, numpy.ndarray]

`save_csvfiles`(*dir=None*)
Move output CSV files 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** (*Optional[str]*) – directory path

Return type None

See also:

stanfit.RunSet.save_csvfiles, *cmdstanpy.from_csv*

`stan_variable`(*var=None*, *inc_warmup=False*)
Return a numpy.ndarray which contains the set of draws for the named Stan program variable. Flattens the chains, leaving the draws in chain order. The first array dimension, corresponds to number of draws or post-warmup draws in the sample, per argument `inc_warmup`. The remaining dimensions correspond to the shape of the Stan program variable.

Underlyingly draws are in chain order, i.e., for a sample with 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 return array has shape (draws X chains, 1).
- If the variable is a vector, the return array has shape (draws X chains, len(vector))
- If the variable is a matrix, the return array has shape (draws X chains, size(dim 1) X size(dim 2))
- If the variable is an array with N dimensions, the return array has shape (draws X chains, size(dim 1) X … X size(dim N))

For example, if the Stan program variable `theta` is a 3x3 matrix, and the sample consists of 4 chains with 1000 post-warmup draws, this function will return a numpy.ndarray with shape (4000,3,3).

Parameters

- **var** (*Optional[str]*) – variable name
- **inc_warmup** (*bool*) – 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`.

Return type numpy.ndarray

See also:

CmdStanMCMC.stan_variables, *CmdStanMLE.stan_variable*, *CmdStanVB.stan_variable*, *CmdStanGQ.stan_variable*

`stan_variables`()
Return a dictionary mapping Stan program variables names to the corresponding numpy.ndarray containing the inferred values.

See also:

CmdStanMCMC.stan_variable, *CmdStanMLE.stan_variables*, *CmdStanVB.stan_variables*, *CmdStanGQ.stan_variables*

Return type Dict[str, numpy.ndarray]

summary(*percentiles=None*, *sig_figs=None*)

Run cmdstan/bin/stansummary over all output CSV files, assemble summary into DataFrame object; first row contains summary statistics for total joint log probability *lp__*, remaining rows contain summary statistics for all parameters, transformed parameters, and generated quantities variables listed in the order in which they were declared in the Stan program.

Parameters

- **percentiles** (*Optional[List[int]]*) – Ordered non-empty list of percentiles to report. Must be integers from (1, 99), inclusive.
- **sig_figs** (*Optional[int]*) – Number of significant figures to report. Must be an integer between 1 and 18. If unspecified, the default precision for the system file I/O is used; the usual value is 6. If precision above 6 is requested, sample must have been produced by CmdStan version 2.25 or later and sampler output precision must equal to or greater than the requested summary precision.

Returns pandas.DataFrame

Return type pandas.core.frame.DataFrame

property chain_ids: List[int]

Chain ids.

property chains: int

Number of chains.

property column_names: Tuple[str, ...]

Names of all outputs from the sampler, comprising sampler parameters and all components of all model parameters, transformed parameters, and quantities of interest. Corresponds to Stan CSV file header row, with names munged to array notation, e.g. *beta[1]* not *beta.1*.

property metadata: *cmdstanpy.stanfit.InferenceMetadata*

Returns object which contains CmdStan configuration as well as information about the names and structure of the inference method and model output variables.

property metric: Optional[numpy.ndarray]

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

property metric_type: Optional[str]

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

property num_draws_sampling: int

Number of sampling (post-warmup) draws per chain, i.e., thinned sampling iterations.

property num_draws_warmup: int

Number of warmup draws per chain, i.e., thinned warmup iterations.

property step_size: Optional[numpy.ndarray]

Step size used by sampler for each chain. When sampler algorithm 'fixed_param' is specified, step size is None.

property thin: int

Period between recorded iterations. (Default is 1).

6.2.3 CmdStanMLE

class cmdstanpy.**CmdStanMLE**(*runset*)

Container for outputs from CmdStan optimization. Created by *CmdStanModel.optimize()*.

Parameters **runset** (cmdstanpy.stanfit.RunSet) –

Return type None

save_csvfiles(*dir=None*)

Move output CSV files 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** (*Optional[str]*) – directory path

Return type None

See also:

stanfit.RunSet.save_csvfiles, *cmdstanpy.from_csv*

stan_variable(*var=None*, *, *inc_iterations=False*, *warn=True*)

Return a numpy.ndarray which contains the estimates for the for the named Stan program variable where the dimensions of the numpy.ndarray match the shape of the Stan program variable.

Parameters

- **var** (*Optional[str]*) – variable name
- **inc_iterations** (*bool*) – When `True` and the intermediate estimates are included in the output, i.e., the optimizer was run with `save_iterations=True`, then intermediate estimates are included. Default value is `False`.
- **warn** (*bool*) –

Return type Union[numpy.ndarray, float]

See also:

CmdStanMLE.stan_variables, *CmdStanMCMC.stan_variable*, *CmdStanVB.stan_variable*, *CmdStanGQ.stan_variable*

stan_variables(*inc_iterations=False*)

Return a dictionary mapping Stan program variables names to the corresponding numpy.ndarray containing the inferred values.

Parameters **inc_iterations** (*bool*) – When `True` and the intermediate estimates are included in the output, i.e., the optimizer was run with `save_iterations=True`, then intermediate estimates are included. Default value is `False`.

Return type Dict[str, Union[numpy.ndarray, float]]

See also:

CmdStanMLE.stan_variable, *CmdStanMCMC.stan_variables*, *CmdStanVB.stan_variables*, *CmdStanGQ.stan_variables*

property column_names: Tuple[str, ...]

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

property metadata: *cmdstanpy.stanfit.InferenceMetadata*

Returns object which contains CmdStan configuration as well as information about the names and structure of the inference method and model output variables.

property optimized_iterations_np: Optional[numpy.ndarray]
Returns all saved iterations from the optimizer and final estimate as a numpy.ndarray which contains all optimizer outputs, i.e., the value for *lp__* as well as all Stan program variables.

property optimized_iterations_pd: Optional[pandas.core.frame.DataFrame]
Returns all saved iterations from the optimizer and final estimate as a pandas.DataFrame which contains all optimizer outputs, i.e., the value for *lp__* as well as all Stan program variables.

property optimized_params_dict: Dict[str, float]
Returns all estimates from the optimizer, including *lp__* as a Python Dict. Only returns estimate from final iteration.

property optimized_params_np: numpy.ndarray
Returns all final estimates from the optimizer as a numpy.ndarray which contains all optimizer outputs, i.e., the value for *lp__* as well as all Stan program variables.

property optimized_params_pd: pandas.core.frame.DataFrame
Returns all final estimates from the optimizer as a pandas.DataFrame which contains all optimizer outputs, i.e., the value for *lp__* as well as all Stan program variables.

6.2.4 CmdStanGQ

class cmdstanpy.**CmdStanGQ**(*runset*, *mcmc_sample*)
Container for outputs from CmdStan generate_quantities run. Created by *CmdStanModel.generate_quantities()*.

Parameters

- **runset** (cmdstanpy.stanfit.RunSet) –
- **mcmc_sample** (cmdstanpy.stanfit.CmdStanMCMC) –

Return type None

draws(*, *inc_warmup=False*, *concat_chains=False*, *inc_sample=False*)
Returns a numpy.ndarray over the generated quantities draws from all chains which is stored column major so that the values for a parameter are contiguous in memory, likewise all draws from a chain are contiguous. By default, returns a 3D array arranged (draws, chains, columns); parameter `concat_chains=True` will return a 2D array where all chains are flattened into a single column, preserving chain order, so that given M chains of N draws, the first N draws are from chain 1, …, and the the last N draws are from chain M.

Parameters

- **inc_warmup** (*bool*) – 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`.
- **concat_chains** (*bool*) – When `True` return a 2D array flattening all all draws from all chains. Default value is `False`.
- **inc_sample** (*bool*) – When `True` include all columns in the mcmc_sample draws array as well, excepting columns for variables already present in the generated quantities drawset. Default value is `False`.

Return type numpy.ndarray

See also:

CmdStanGQ.draws_pd, *CmdStanGQ.draws_xr*, *CmdStanMCMC.draws*

draws_pd(*vars=None*, *inc_warmup=False*, *inc_sample=False*)

Returns the generated quantities draws as a pandas DataFrame. Flattens all chains into single column. Container variables (array, vector, matrix) will span multiple columns, one column per element. E.g. variable 'matrix[2,2] foo' spans 4 columns: 'foo[1,1], ... foo[2,2]'.

Parameters

- **vars** (*Optional[Union[List[str], str]]*) – optional list of variable names.
- **inc_warmup** (*bool*) – 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`.
- **inc_sample** (*bool*) –

Return type pandas.core.frame.DataFrame

See also:

CmdStanGQ.draws, *CmdStanGQ.draws_xr*, *CmdStanMCMC.draws_pd*

draws_xr(*vars=None*, *inc_warmup=False*, *inc_sample=False*)

Returns the generated quantities draws as a xarray Dataset.

Parameters

- **vars** (*Optional[Union[List[str], str]]*) – optional list of variable names.
- **inc_warmup** (*bool*) – When `True` and the warmup draws are present in the MCMC sample, then the warmup draws are included. Default value is `False`.
- **inc_sample** (*bool*) –

Return type xarray.core.dataset.Dataset

See also:

CmdStanGQ.draws, *CmdStanGQ.draws_pd*, *CmdStanMCMC.draws_xr*

save_csvfiles(*dir=None*)

Move output CSV files 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** (*Optional[str]*) – directory path

Return type None

See also:

stanfit.RunSet.save_csvfiles, *cmdstanpy.from_csv*

stan_variable(*var=None*, *inc_warmup=False*)

Return a numpy.ndarray which contains the set of draws for the named Stan program variable. Flattens the chains, leaving the draws in chain order. The first array dimension, corresponds to number of draws in the sample. The remaining dimensions correspond to the shape of the Stan program variable.

Underlyingly draws are in chain order, i.e., for a sample with 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 return array has shape (draws X chains, 1).
- If the variable is a vector, the return array has shape (draws X chains, len(vector))
- If the variable is a matrix, the return array has shape (draws X chains, size(dim 1) X size(dim 2))
- If the variable is an array with N dimensions, the return array has shape (draws X chains, size(dim 1) X ... X size(dim N))

For example, if the Stan program variable `theta` is a 3x3 matrix, and the sample consists of 4 chains with 1000 post-warmup draws, this function will return a numpy.ndarray with shape (4000,3,3).

Parameters

- **var** (*Optional[str]*) – variable name
- **inc_warmup** (*bool*) – When `True` and the warmup draws are present in the MCMC sample, then the warmup draws are included. Default value is `False`.

Return type numpy.ndarray

See also:

CmdStanGQ.stan_variables, *CmdStanMCMC.stan_variable*, *CmdStanMLE.stan_variable*, *CmdStanVB.stan_variable*

stan_variables(*inc_warmup=False*)

Return a dictionary mapping Stan program variables names to the corresponding numpy.ndarray containing the inferred values.

Parameters **inc_warmup** (*bool*) – When `True` and the warmup draws are present in the MCMC sample, then the warmup draws are included. Default value is `False`

Return type Dict[str, numpy.ndarray]

See also:

CmdStanGQ.stan_variable, *CmdStanMCMC.stan_variables*, *CmdStanMLE.stan_variables*, *CmdStanVB.stan_variables*

property chain_ids: List[int]

Chain ids.

property chains: int

Number of chains.

property column_names: Tuple[str, ...]

Names of generated quantities of interest.

property metadata: *cmdstanpy.stanfit.InferenceMetadata*

Returns object which contains CmdStan configuration as well as information about the names and structure of the inference method and model output variables.

6.2.5 CmdStanVB

class cmdstanpy.**CmdStanVB**(*runset*)

Container for outputs from CmdStan variational run. Created by *CmdStanModel.variational()*.

Parameters **runset** (cmdstanpy.stanfit.RunSet) –

Return type None

save_csvfiles(*dir=None*)

Move output CSV files 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** (*Optional[str]*) – directory path

Return type None

See also:

stanfit.RunSet.save_csvfiles, *cmdstanpy.from_csv*

stan_variable(*var=None*)

Return a numpy.ndarray which contains the estimates for the for the named Stan program variable where the dimensions of the numpy.ndarray match the shape of the Stan program variable.

Parameters **var** (*Optional[str]*) – variable name

Return type Union[numpy.ndarray, float]

See also:

CmdStanVB.stan_variables, *CmdStanMCMC.stan_variable*, *CmdStanMLE.stan_variable*, *CmdStanGQ.stan_variable*

stan_variables()

Return a dictionary mapping Stan program variables names to the corresponding numpy.ndarray containing the inferred values.

See also:

CmdStanVB.stan_variable, *CmdStanMCMC.stan_variables*, *CmdStanMLE.stan_variables*, *CmdStanGQ.stan_variables*

Return type Dict[str, Union[numpy.ndarray, float]]

property column_names: Tuple[str, ...]

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

property columns: int

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

property eta: float

Step size scaling parameter 'eta'

property metadata: *cmdstanpy.stanfit.InferenceMetadata*

Returns object which contains CmdStan configuration as well as information about the names and structure of the inference method and model output variables.

property variational_params_dict: Dict[str, numpy.ndarray]

Returns inferred parameter means as Dict.

property variational_params_np: numpy.ndarray

Returns inferred parameter means as numpy array.

property variational_params_pd: pandas.core.frame.DataFrame

Returns inferred parameter means as pandas DataFrame.

property variational_sample: numpy.ndarray

Returns the set of approximate posterior output draws.

6.3 Functions

6.3.1 show_versions

cmdstanpy.**show_versions**(*output=True*)

Prints out system and dependency information for debugging

Parameters **output** (*bool*) –

Return type str

6.3.2 cmdstan_path

cmdstanpy.**cmdstan_path**()

Validate, then return CmdStan directory path.

Return type str

6.3.3 install_cmdstan

cmdstanpy.**install_cmdstan**(*version=None*, *dir=None*, *overwrite=False*, *compiler=False*, *progress=False*, *verbose=False*, *cores=1*)

Download and install a CmdStan release from GitHub. Downloads the release tar.gz file to temporary storage. Retries GitHub requests in order to allow for transient network outages. Builds CmdStan executables and tests the compiler by building example model `bernoulli.stan`.

Parameters

- **version** (*Optional[str]*) – CmdStan version string, e.g. "2.24.1". Defaults to latest CmdStan release.
- **dir** (*Optional[str]*) – Path to install directory. Defaults to hidden directory `$HOME/.cmdstan`. If no directory is specified and the above directory does not exist, directory `$HOME/.cmdstan` will be created and populated.
- **overwrite** (*bool*) – Boolean value; when `True`, will overwrite and rebuild an existing CmdStan installation. Default is `False`.
- **compiler** (*bool*) – Boolean value; when `True` on WINDOWS ONLY, use the C++ compiler from the `install_cxx_toolchain` command or install one if none is found.
- **progress** (*bool*) – Boolean value; when `True`, show a progress bar for downloading and unpacking CmdStan. Default is `False`.
- **verbose** (*bool*) – Boolean value; when `True`, show console output from all intallation steps, i.e., download, build, and test CmdStan release. Default is `False`.
- **cores** (*int*) – Integer, number of cores to use in the `make` command. Default is 1 core.

Returns Boolean value; `True` for success.

Return type bool

6.3.4 set_cmdstan_path

cmdstanpy.**set_cmdstan_path**(*path*)

Validate, then set CmdStan directory path.

Parameters **path** (*str*) –

Return type None

6.3.5 cmdstan_version

cmdstanpy.**cmdstan_version**()

Parses version string out of CmdStan makefile variable CMDSTAN_VERSION, returns Tuple(Major, minor).

If CmdStan installation is not found or cannot parse version from makefile logs warning and returns None. Lenient behavoir required for CI tests, per comment: https://github.com/stan-dev/cmdstanpy/pull/321#issuecomment-733817554

Return type Optional[Tuple[int, ...]]

6.3.6 set_make_env

cmdstanpy.**set_make_env**(*make*)

set MAKE environmental variable.

Parameters **make** (*str*) –

Return type None

6.3.7 from_csv

cmdstanpy.**from_csv**(*path=None*, *method=None*)

Instantiate a CmdStan object from a the Stan CSV files from a CmdStan run. CSV files are specified from either a list of Stan CSV files or a single filepath which can be either a directory name, a Stan CSV filename, or a pathname pattern (i.e., a Python glob). The optional argument 'method' checks that the CSV files were produced by that method. Stan CSV files from CmdStan methods 'sample', 'optimize', and 'variational' result in objects of class CmdStanMCMC, CmdStanMLE, and CmdStanVB, respectively.

Parameters

- **path** (*Optional[Union[List[str], str]]*) – directory path
- **method** (*Optional[str]*) – method name (optional)

Returns either a CmdStanMCMC, CmdStanMLE, or CmdStanVB object

Return type Optional[Union[*cmdstanpy.stanfit.CmdStanMCMC*, *cmdstanpy.stanfit.CmdStanMLE*, *cmdstanpy.stanfit.CmdStanVB*]]

6.3.8 write_stan_json

cmdstanpy.**write_stan_json**(*path*, *data*)

Dump a mapping of strings to data to a JSON file.

Values can be any numeric type, a boolean (converted to int), or any collection compatible with `numpy.asarray()`, e.g a `pandas.Series`.

Produces a file compatible with the Json Format for Cmdstan

Parameters

- **path** (*str*) – File path for the created json. Will be overwritten if already in existence.
- **data** (*Mapping[str, Any]*) – A mapping from strings to values. This can be a dictionary or something more exotic like an `xarray.Dataset`. This will be copied before type conversion, not modified

Return type None

genindex

PYTHON MODULE INDEX

INDEX

M

N

O

P

R

S

T

U

V

W