CmdStanPy Documentation

Release 1.0.7

Stan Development Team

Aug 25, 2022

CONTENTS

1 Installation **3**
- 1.1 Conda: install CmdStanPy, CmdStan, C++ toolchain . . . 3
- 1.2 PyPI: install package CmdStanPy . . . 4
- 1.3 GitHub: install from the CmdStanPy repository . . . 4
- 1.4 CmdStan Installation . . . 4
  - 1.4.1 C++ Toolchain Requirements . . . 5
  - 1.4.2 Function `install_cmdstan` . . . 5
  - 1.4.3 DIY Installation . . . 6
- 1.5 Locating the CmdStan installation directory . . . 6

2 User's Guide **7**
- 2.1 Overview . . . 7
- 2.2 "Hello, World!" . . . 7
  - 2.2.1 Fitting a Stan model using the NUTS-HMC sampler . . . 7
    - 2.2.1.1 The Stan model . . . 8
    - 2.2.1.2 Data inputs . . . 9
    - 2.2.1.3 Fitting the model . . . 9
    - 2.2.1.4 Accessing the results . . . 9
    - 2.2.1.5 CmdStan utilities: `stansummary`, `diagnose` . . . 12
- 2.3 CmdStanPy Workflow . . . 12
  - 2.3.1 Compile the Stan model . . . 13
  - 2.3.2 Assemble input and initialization data . . . 14
  - 2.3.3 Run the CmdStan inference engine . . . 14
  - 2.3.4 Validate, view, export the inference engine outputs . . . 14
    - 2.3.4.1 Metadata . . . 15
    - 2.3.4.2 Output data . . . 15
- 2.4 Controlling Outputs . . . 15
  - 2.4.1 CSV File Outputs . . . 15
  - 2.4.2 Logging . . . 17
- 2.5 CmdStanPy Examples . . . 20
  - 2.5.1 MCMC Sampling . . . 20
    - 2.5.1.1 Overview . . . 20
    - 2.5.1.2 Fitting the model and data . . . 21
    - 2.5.1.3 Checking the fit . . . 26
    - 2.5.1.4 Accessing the sampler outputs . . . 29
    - 2.5.1.5 Saving the sampler output files . . . 32
    - 2.5.1.6 Parallelization via multi-threaded processing . . . 32
  - 2.5.2 Maximum Likelihood Estimation . . . 35
  - 2.5.3 Variational Inference in Stan . . . 35
    - 2.5.3.1 Example: variational inference for model `bernoulli.stan` . . . 36

2.5.4 Using Variational Estimates to Initialize the NUTS-HMC Sampler . . . 38
2.5.4.1 Model and data . . . 38
2.5.4.2 Run Stan's variational inference algorithm, obtain fitted estimates . . . 39
2.5.5 Generating new quantities of interest. . . . 42
2.5.5.1 Example: add posterior predictive checks to `bernoulli.stan` . . . 42
2.5.6 Advanced Topic: Using External C++ Functions . . . 46

3 API Reference **49**
3.1 Internal API Reference . . . 49
3.1.1 Classes . . . 49
3.1.1.1 InferenceMetadata . . . 49
3.1.1.2 RunSet . . . 50
3.1.1.3 CompilerOptions . . . 51
3.1.1.4 CmdStanArgs . . . 52
3.1.1.5 SamplerArgs . . . 53
3.1.1.6 OptimizeArgs . . . 54
3.1.1.7 VariationalArgs . . . 55
3.2 Classes . . . 56
3.2.1 CmdStanModel . . . 56
3.2.2 CmdStanMCMC . . . 65
3.2.3 CmdStanMLE . . . 69
3.2.4 CmdStanGQ . . . 70
3.2.5 CmdStanVB . . . 73
3.3 Functions . . . 74
3.3.1 show_versions . . . 74
3.3.2 cmdstan_path . . . 74
3.3.3 install_cmdstan . . . 74
3.3.4 set_cmdstan_path . . . 75
3.3.5 cmdstan_version . . . 75
3.3.6 set_make_env . . . 75
3.3.7 from_csv . . . 76
3.3.8 write_stan_json . . . 76

4 What's New **77**
4.1 CmdStanPy 1.0.6 . . . 77
4.2 CmdStanPy 1.0.6 . . . 77
4.3 CmdStanPy 1.0.5 . . . 77
4.4 CmdStanPy 1.0.4 . . . 77
4.5 CmdStanPy 1.0.3 . . . 78
4.6 CmdStanPy 1.0.2 . . . 78
4.7 CmdStanPy 1.0.1 . . . 78
4.8 CmdStanPy 1.0.0 . . . 79

5 Community **81**
5.1 Project templates . . . 81
5.2 Software . . . 81

Python Module Index **83**

Index **85**

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

INSTALLATION

CmdStanPy is a pure-Python3 package which wraps CmdStan, the command-line interface to Stan which is written in C++. Therefore, in addition to Python3, CmdStanPy requires a modern C++ toolchain in order to build and run Stan models. There are several ways to install CmdStanPy and the underlying CmdStan components.

- You can download CmdStanPy, CmdStan, and the C++ toolchain from conda-forge.
- You can download the CmdStanPy package from PyPI using pip.
- If you want the current development version, you can clone the GitHub CmdStanPy repository.

If you install CmdStanPy from PyPI or GitHub you will need to install CmdStan as well, see section *CmdStan Installation* below.

1.1 Conda: install CmdStanPy, CmdStan, C++ toolchain

If you use conda, you can install CmdStanPy and the underlying CmdStan components from the conda-forge repository via the following command:

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

This command creates a new conda environment named `stan` and downloads and installs the `cmdstanpy` package as well as CmdStan and the required C++ toolchain.

To install into an existing environment, use the conda `install` command instead of `create`:

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

Whichever installation method you use, afterwards you must activate the new environment or deactivate/activate the existing one. For example, if you installed cmdstanpy into a new environment `stan`, run the command

```
conda activate stan
```

By default, the latest release of CmdStan is installed. If you require a specific release of CmdStan, CmdStan versions 2.26.1 and *newer* can be installed by specifying `cmdstan==VERSION` in the install command. Versions before 2.26.1 are not available from conda but can be downloaded from the CmdStan releases page.

A Conda environment is a directory that contains a specific collection of Conda packages. To see the locations of your conda environments, use the command

```
conda info -e
```

The shell environment variable `CONDA_PREFIX` points to the active conda environment (if any). Both CmdStan and the C++ toolchain are installed into the `bin` subdirectory of the conda environment directory, i.e., `$CONDA_PREFIX/bin/cmdstan` (Linux, MacOS), `%CONDA_PREFIX%\bin\cmdstan` (Windows).

1.2 PyPI: install package CmdStanPy

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]
```

1.3 GitHub: install from the CmdStanPy repository

To install the current develop branch from GitHub:

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

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: If you intend to run CmdStanPy from within a Jupyter notebook, you may need to install the ipywidgets. This will allow for progress bars implemented using the tqdm to display properly in the browser. For further help on Jupyter notebook installation and configuration , see ipywidgets installation instructions and this tqdm GitHub issue.

1.4 CmdStan Installation

If you have installed CmdStanPy from PyPI or Github, **you must install CmdStan**. The recommended way to do so is via the `install_cmdstan` function *described below*.

If you installed CmdStanPy with conda, CmdStan and the C++ toolchain, both CmdStan and the C++ toolchain are installed into directory `$CONDA_PREFIX/bin` and you don't need to do any further installs.

1.4.1 C++ Toolchain Requirements

To compile a Stan program requires a modern C++ compiler and the GNU-Make build utility. These vary by operating system.

Linux The required C++ compiler is `g++ 4.9 3`. On most systems the GNU-Make utility is pre-installed and is the default `make` utility. There is usually a pre-installed C++ compiler as well, but not necessarily new enough.

MacOS The Xcode and Xcode command line tools must be installed. Xcode is available for free from the Mac App Store. To install the Xcode command line tools, run the shell command: `xcode-select --install`.

Windows We recommend using the RTools 4.0 toolchain which contains a `g++ 8` compiler and `Mingw`, the native Windows equivalent of the GNU-Make utility. This can be installed allong with CmdStan when you invoke the function *cmdstanpy.install_cmdstan()* with argument `compiler=True`.

1.4.2 Function `install_cmdstan`

CmdStanPy provides the function *cmdstanpy.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. On Windows, the `compiler` option will install the C++ toolchain.

- From Python

```
import cmdstanpy
cmdstanpy.install_cmdstan()
cmdstanpy.install_cmdstan(compiler=True)  # only valid on Windows
```

- From the command line on Linux or MacOSX

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

- On Windows

```
install_cmdstan --compiler
dir "%HOME%/.cmdstan"
```

The argument `--interactive` (or `-i`) can be used to run the installation script in an interactive prompt. This will ask you about the various options to the installation script, with reasonable defaults set for all questions.

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
```

1.4.3 DIY Installation

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

1.5 Locating the CmdStan installation directory

CmdStanPy uses the environment variable `CMDSTAN` to register the CmdStan installation location.

- If you use conda to install CmdStanPy, CmdStan is installed into location `$CONDA_PREFIX/bin/cmdstan` (Linux, MacOS), `%CONDA_PREFIX%\bin\cmdstan` (Windows) and the environment variable `CMDSTAN` is set accordingly.
- If no environment variable `CMDSTAN` is set, CmdStanPy will try to locate a CmdStan installation in the default install location, which is a directory named `.cmdstan` in your `$HOME` directory.

If you have installed CmdStan from a GitHub release or by cloning the CmdStan repository, you will need to set this location, either via the `CMDSTAN` environment variable, or via the CmdStanPy command `set_cmdstan_path`

```
from cmdstanpy import cmdstan_path, set_cmdstan_path

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

CHAPTER

TWO

USER'S GUIDE

This section provides examples and some narrative description about the usage of CmdStanPy.

2.1 Overview

CmdStanPy is a lightweight interface to Stan for Python. It provides a small set of classes and methods for doing Bayesian inference given a probability model and data. With CmdStanPy, you can:

- Compile a Stan model.
- Do inference on the model conditioned on the data, using one of Stan inference algorithms
  - Exact Bayesian estimation using the NUTS-HMC sampler.
  - Approximate Bayesian estimation using ADVI.
  - MAP estimation by optimization.
- Generate new quantities of interest from a model given an existing sample.
- Manage the resulting inference engine outputs: extract information, summarize results, and save the outputs.

CmdStanPy wraps the CmdStan file-based command line interface. It is lightweight in that it uses minimal memory beyond the memory used by CmdStan, thus CmdStanPy has the potential to fit more complex models to larger datasets than might be possible in PyStan or RStan.

CmdStanPy is designed to support the development, testing, and deployment of a Stan model. CmdStanPy manages the Stan program files, data files, and CmdStan output files. By default, output files are written to a temporary filesystem which persists throughout the session. This is appropriate behavior during model development because it allows the user to test many models without filsystem clutter or worse. Once deployed into production, the user can specify the output directory for the CmdStan outputs.

2.2 "Hello, World!"

2.2.1 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`. The `bernoulli.stan` is a Hello, World! program which illustrates the basic syntax of the Stan language. It 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 Stan User's Guide.

2.2.1.1 The Stan model

The model `bernoulli.stan` is a trivial model: 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. A *CmdStanModel* can be instantiated from a Stan file or its corresponding compiled executable file.

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

In [2]: from cmdstanpy import CmdStanModel

# specify Stan program file
In [3]: stan_file = os.path.join('users-guide', 'examples', 'bernoulli.stan')

# instantiate the model object
In [4]: model = CmdStanModel(stan_file=stan_file)

# inspect model object
In [5]: print(model)
CmdStanModel: name=bernoulli
         stan_file=/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/
↪v1.0.7/docsrc/users-guide/examples/bernoulli.stan
         exe_file=/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/
↪v1.0.7/docsrc/users-guide/examples/bernoulli
         compiler_options=stanc_options={}, cpp_options={}

# inspect compiled model
In [6]: print(model.exe_info())
{'stan_version_major': '2', 'stan_version_minor': '30', 'stan_version_patch': '0', 'STAN_
↪THREADS': 'false', 'STAN_MPI': 'false', 'STAN_OPENCL': 'false', 'STAN_NO_RANGE_CHECKS':
↪ 'false', 'STAN_CPP_OPTIMS': 'false'}
```

2.2.1.2 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]
}
```

2.2.1.3 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.

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

# fit the model
In [8]: fit = model.sample(data=data_file)
INFO:cmdstanpy:CmdStan start processing
                                                                                      ␣
↪                                                                                     ␣
↪                                                                                     ␣
↪
INFO:cmdstanpy:CmdStan done processing.
```

Note this model can be fit using other methods

- the *variational()* method does approximate Bayesian inference and returns a *CmdStanVB* object
- the *optimize()* method does maximum likelihood estimation and returns a *CmdStanMLE* object

2.2.1.4 Accessing the results

The sampler outputs are the set of per-chain Stan CSV files, a non-standard CSV file format. Each data row of the Stan CSV file contains the per-iteration estimate of the Stan model parameters, transformed parameters, and generated quantities variables. Container variables, i.e., vector, row-vector, matrix, and array variables are necessarily serialized into a single row's worth of data. The output objects parse the set of Stan CSV files into a set of in-memory data structures and provide accessor functions for the all estimates and metadata. CmdStanPy makes a distinction between the per-iteration model outputs and the per-iteration algorithm outputs: the former are 'stan_variables' and the latter are 'method_variables'.

The *CmdStanMCMC* object provides the following accessor methods:

- *stan_variable()*: returns an numpy.ndarray whose structure corresponds to the Stan program variable structure
- *stan_variables()*: returns an Python dictionary mapping the Stan program variable names to the corresponding numpy.ndarray.

- *draws()*: returns a numpy.ndarray which is either a 3-D array draws X chains X CSV columns, or a 2-D array draws X columns, where the chains are concatenated into a single column. The argument *vars* can be used to restrict this to just the columns for one or more variables.
- *draws_pd()*: returns a pandas.DataFrame over all columns in the Stan CSV file. The argument *vars* can be used to restrict this to one or more variables.
- *draws_xr()*: returns an xarray.Dataset which maps model variable names to their respective values. The argument *vars* can be used to restrict this to one or more variables.
- *method_variables()*: returns a Python dictionary over the sampler diagnostic/information output columns which by convention end in __, e.g., lp__.

```
# access model variable by name
In [9]: print(fit.stan_variable('theta'))
[0.352324 0.409213 0.429472 ... 0.495313 0.170008 0.205309]

In [10]: print(fit.draws_pd('theta')[:3])
      theta
0  0.352324
1  0.409213
2  0.429472

In [11]: print(fit.draws_xr('theta'))
<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 0.3523 0.4092 0.4295 ... 0.4953 0.17 0.2053
Attributes:
    stan_version:        2.30.0
    model:               bernoulli_model
    num_draws_sampling:  1000

# access all model variables
In [12]: for k, v in fit.stan_variables().items():
   ....:     print(f'{k}\t{v.shape}')
   ....:
theta        (4000,)

# access the sampler method variables
In [13]: for k, v in fit.method_variables().items():
   ....:     print(f'{k}\t{v.shape}')
   ....:
lp__         (1000, 4)
accept_stat__        (1000, 4)
stepsize__        (1000, 4)
treedepth__        (1000, 4)
n_leapfrog__        (1000, 4)
divergent__        (1000, 4)
energy__        (1000, 4)

# access all Stan CSV file columns
```

(continues on next page)

(continued from previous page)

```
In [14]: print(f'numpy.ndarray of draws: {fit.draws().shape}')
numpy.ndarray of draws: (1000, 4, 8)

In [15]: fit.draws_pd()
Out[15]:
         lp__  accept_stat__  stepsize__  ...  divergent__  energy__     theta
0    -7.03889       0.287212      1.0417  ...          0.0   8.69639  0.352324
1    -7.41725       0.852316      1.0417  ...          0.0   7.50872  0.409213
2    -7.58633       0.928636      1.0417  ...          0.0   7.85016  0.429472
3    -6.98162       0.997829      1.0417  ...          0.0   7.84039  0.171673
4    -6.82514       1.000000      1.0417  ...          0.0   6.94285  0.203175
...       ...            ...         ...  ...          ...       ...       ...
3995 -6.76187       0.995207      1.1115  ...          0.0   6.85153  0.229594
3996 -6.83718       0.978261      1.1115  ...          0.0   6.87352  0.305082
3997 -8.26204       0.575760      1.1115  ...          0.0   8.26245  0.495313
3998 -6.99278       0.865419      1.1115  ...          0.0   9.15931  0.170008
3999 -6.81793       0.943203      1.1115  ...          0.0   7.65179  0.205309

[4000 rows x 8 columns]
```

In addition to the MCMC sample itself, the CmdStanMCMC object provides access to the the per-chain HMC tuning parameters from the NUTS-HMC adaptive sampler, (if present).

```
In [16]: print(fit.metric_type)
diag_e

In [17]: print(fit.metric)
[[0.552597]
 [0.494052]
 [0.457823]
 [0.454162]]

In [18]: print(fit.step_size)
[1.0417   0.920896 0.971847 1.1115  ]
```

The CmdStanMCMC object also provides access to metadata about the model and the sampler run.

```
In [19]: print(fit.metadata.cmdstan_config['model'])
bernoulli_model

In [20]: print(fit.metadata.cmdstan_config['seed'])
71356

In [21]: print(fit.metadata.stan_vars_cols.keys())
dict_keys(['theta'])

In [22]: print(fit.metadata.method_vars_cols.keys())
dict_keys(['lp__', 'accept_stat__', 'stepsize__', 'treedepth__', 'n_leapfrog__',
↪'divergent__', 'energy__'])
```

2.2.1.5 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 [23]: fit.summary()
Out[23]:
          Mean     MCSE    StdDev  ...    N_Eff  N_Eff/s    R_hat
lp__  -7.26173  0.01828  0.718175  ...  1543.55  20580.6  1.00246
theta  0.25479  0.00336  0.119546  ...  1265.82  16877.6  1.00296

[2 rows x 9 columns]
```

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 [24]: print(fit.diagnose())
Processing csv files: /tmp/tmpwxwxvhm2/bernoulli8vmgytsw/bernoulli-20220825133649_1.csv,␣
↪/tmp/tmpwxwxvhm2/bernoulli8vmgytsw/bernoulli-20220825133649_2.csv, /tmp/tmpwxwxvhm2/
↪bernoulli8vmgytsw/bernoulli-20220825133649_3.csv, /tmp/tmpwxwxvhm2/bernoulli8vmgytsw/
↪bernoulli-20220825133649_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.
```

2.3 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 filesys-

tem 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.

2.3.1 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
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

```

(continues on next page)

(continued from previous page)

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

2.3.2 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.

2.3.3 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.

2.3.4 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

2.3.4.1 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.

2.3.4.2 Output data

The resulting Stan CSV file or set of files are assembled into an inference result object.

- *CmdStanMCMC* object contains the *sample()* outputs
- *CmdStanVB* object contains the *variational()* outputs
- *CmdStanMLE* object contains the *optimize()* outputs
- *CmdStanGQ* object contains the *generate_quantities()* outputs

The objects provide accessor methods which return this information either as tabular data (i.e., in terms of the per-chain CSV file rows and columns), or as structured objects which correspond to the variables in the Stan model and the individual diagnostics produced by the inference method.

The `stan_variables` method returns a Python dict over all Stan model variables, see *stan_variables()*.

The `stan_variable` method returns a single model variable as a numpy.ndarray object with the same structure (per draw) as the Stan program variable, see *stan_variable()*.

The `method_variables` method returns a Python dict over all inference method variables, cf *method_variables()*

The output from the methods *CmdStanMCMC* and *CmdStanGQ* return the sample contents in tabular form, see *draws()* and *draws_pd()*. Similarly, the *draws_xr()* method returns the sample contents as an `xarray.Dataset` which is a mapping from variable names to their respective values.

2.4 Controlling Outputs

2.4.1 CSV File Outputs

Underlyingly, the CmdStan outputs are a set of per-chain Stan CSV files. The filenames follow the template '<model_name>-<YYYYMMDDHHMMSS>-<chain_id>' plus the file suffix '.csv'. CmdStanPy also captures the per-chain console and error messages.

```
In [1]: import os

In [2]: from cmdstanpy import CmdStanModel

In [3]: stan_file = os.path.join('users-guide', 'examples', 'bernoulli.stan')

```

(continues on next page)

(continued from previous page)

```
In [4]: model = CmdStanModel(stan_file=stan_file)

In [5]: data_file = os.path.join('users-guide', 'examples', 'bernoulli.data.json')

In [6]: fit = model.sample(data=data_file)
INFO:cmdstanpy:CmdStan start processing

↪                                                                                ␣
↪                                                                                ␣
↪                                                                                ␣
INFO:cmdstanpy:CmdStan done processing.

# printing the object reports sampler commands, output files
In [7]: print(fit)
CmdStanMCMC: model=bernoulli chains=4['method=sample', 'algorithm=hmc', 'adapt',
↪'engaged=1']
 csv_files:
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_1.csv
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_2.csv
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_3.csv
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_4.csv
 output_files:
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_0-stdout.txt
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_1-stdout.txt
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_2-stdout.txt
        /tmp/tmpwxwxvhm2/bernoulli8i689zpv/bernoulli-20220825133650_3-stdout.txt
```

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.

```
In [8]: fit = model.sample(data=data_file, output_dir="./outputs/")
INFO:cmdstanpy:created output directory: /home/docs/checkouts/readthedocs.org/user_
↪builds/cmdstanpy/checkouts/v1.0.7/docsrc/outputs
INFO:cmdstanpy:CmdStan start processing

↪                                                                                ␣
↪                                                                                ␣
↪                                                                                ␣
INFO:cmdstanpy:CmdStan done processing.

In [9]: !ls outputs/
bernoulli-20220825133650_0-stdout.txt  bernoulli-20220825133650_2.csv
bernoulli-20220825133650_1-stdout.txt  bernoulli-20220825133650_3-stdout.txt
bernoulli-20220825133650_1.csv               bernoulli-20220825133650_3.csv
bernoulli-20220825133650_2-stdout.txt  bernoulli-20220825133650_4.csv
```

Alternatively, the *save_csvfiles()* function moves the CSV files to a specified directory.

```
In [10]: fit = model.sample(data=data_file)
INFO:cmdstanpy:CmdStan start processing

↪                                                                                ␣
```

(continues on next page)

```
↪                                                                                ␣
↪
```

(continued from previous page)

```
INFO:cmdstanpy:CmdStan done processing.

In [11]: fit.save_csvfiles(dir='some/path')

In [12]: !ls some/path
bernoulli-20220825133651_1.csv        bernoulli-20220825133651_3.csv
bernoulli-20220825133651_2.csv        bernoulli-20220825133651_4.csv
```

2.4.2 Logging

You may notice CmdStanPy can produce a lot of output when it is running:

```
In [13]: fit = model.sample(data=data_file, show_progress=False)
INFO:cmdstanpy:CmdStan start processing
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [2] start processing
INFO:cmdstanpy:Chain [1] done processing
INFO:cmdstanpy:Chain [3] start processing
INFO:cmdstanpy:Chain [2] done processing
INFO:cmdstanpy:Chain [4] start processing
INFO:cmdstanpy:Chain [3] done processing
INFO:cmdstanpy:Chain [4] done processing
```

This output is managed through the built-in `logging` module. For example, it can be disabled entirely:

```
In [14]: import logging

In [15]: cmdstanpy_logger = logging.getLogger("cmdstanpy")

In [16]: cmdstanpy_logger.disabled = True

# look, no output!
In [17]: fit = model.sample(data=data_file, show_progress=False)
```

Or one can remove the logging handler that CmdStanPy installs by default and install their own for more fine-grained control. For example, the following code sends all logs (including the `DEBUG` logs, which are hidden by default), to a file.

DEBUG logging is useful primarily to developers or when trying to hunt down an issue.

```
In [18]: cmdstanpy_logger.disabled = False

# remove all existing handlers
In [19]: cmdstanpy_logger.handlers = []

In [20]: cmdstanpy_logger.setLevel(logging.DEBUG)

In [21]: handler = logging.FileHandler('all.log')

In [22]: handler.setLevel(logging.DEBUG)

In [23]: handler.setFormatter(
```

(continues on next page)

(continued from previous page)

```
   ....:     logging.Formatter(
   ....:         '%(asctime)s - %(name)s - %(levelname)s - %(message)s',
   ....:         "%H:%M:%S",
   ....:     )
   ....: )
   ....:

In [24]: cmdstanpy_logger.addHandler(handler)
```

Now, if we run the model and check the contents of the file, we will see all the possible logging.

```
In [25]: fit = model.sample(data=data_file, show_progress=False)

In [26]: with open('all.log','r') as logs:
   ....:     for line in logs.readlines():
   ....:         print(line.strip())
   ....:
13:36:52 - cmdstanpy - DEBUG - cmd: /home/docs/checkouts/readthedocs.org/user_builds/
↪cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/examples/bernoulli info
cwd: None
13:36:52 - cmdstanpy - INFO - CmdStan start processing
13:36:52 - cmdstanpy - DEBUG - idx 0
13:36:52 - cmdstanpy - DEBUG - idx 1
13:36:52 - cmdstanpy - DEBUG - running CmdStan, num_threads: 1
13:36:52 - cmdstanpy - DEBUG - running CmdStan, num_threads: 1
13:36:52 - cmdstanpy - DEBUG - CmdStan args: ['/home/docs/checkouts/readthedocs.org/user_
↪builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/examples/bernoulli', 'id=1',
↪'random', 'seed=68594', 'data', 'file=users-guide/examples/bernoulli.data.json',
↪'output', 'file=/tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_1.csv',
↪'method=sample', 'algorithm=hmc', 'adapt', 'engaged=1']
13:36:52 - cmdstanpy - DEBUG - CmdStan args: ['/home/docs/checkouts/readthedocs.org/user_
↪builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/examples/bernoulli', 'id=2',
↪'random', 'seed=68594', 'data', 'file=users-guide/examples/bernoulli.data.json',
↪'output', 'file=/tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_2.csv',
↪'method=sample', 'algorithm=hmc', 'adapt', 'engaged=1']
13:36:52 - cmdstanpy - INFO - Chain [1] start processing
13:36:52 - cmdstanpy - INFO - Chain [2] start processing
13:36:52 - cmdstanpy - INFO - Chain [2] done processing
13:36:52 - cmdstanpy - INFO - Chain [1] done processing
13:36:52 - cmdstanpy - DEBUG - idx 2
13:36:52 - cmdstanpy - DEBUG - running CmdStan, num_threads: 1
13:36:52 - cmdstanpy - DEBUG - idx 3
13:36:52 - cmdstanpy - DEBUG - running CmdStan, num_threads: 1
13:36:52 - cmdstanpy - DEBUG - CmdStan args: ['/home/docs/checkouts/readthedocs.org/user_
↪builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/examples/bernoulli', 'id=4',
↪'random', 'seed=68594', 'data', 'file=users-guide/examples/bernoulli.data.json',
↪'output', 'file=/tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_4.csv',
↪'method=sample', 'algorithm=hmc', 'adapt', 'engaged=1']
13:36:52 - cmdstanpy - INFO - Chain [4] start processing
13:36:52 - cmdstanpy - DEBUG - CmdStan args: ['/home/docs/checkouts/readthedocs.org/user_
↪builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/examples/bernoulli', 'id=3',
↪'random', 'seed=68594', 'data', 'file=users-guide/examples/bernoulli.data.json',
↪'output', 'file=/tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_3.csv',
↪'method=sample', 'algorithm=hmc', 'adapt', 'engaged=1']
```

(continues on next page)

(continued from previous page)

```
13:36:52 - cmdstanpy - INFO - Chain [3] start processing
13:36:52 - cmdstanpy - INFO - Chain [4] done processing
13:36:52 - cmdstanpy - INFO - Chain [3] done processing
13:36:52 - cmdstanpy - DEBUG - runset
RunSet: chains=4, chain_ids=[1, 2, 3, 4], num_processes=4
cmd (chain 1):
['/home/docs/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.7/docsrc/
↪users-guide/examples/bernoulli', 'id=1', 'random', 'seed=68594', 'data', 'file=users-
↪guide/examples/bernoulli.data.json', 'output', 'file=/tmp/tmpwxwxvhm2/
↪bernoullirgl40flg/bernoulli-20220825133652_1.csv', 'method=sample', 'algorithm=hmc',
↪'adapt', 'engaged=1']
retcodes=[0, 0, 0, 0]
per-chain output files (showing chain 1 only):
csv_file:
/tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_1.csv
console_msgs (if any):
/tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_0-stdout.txt
13:36:52 - cmdstanpy - DEBUG - Chain 1 console:
method = sample (Default)
sample
num_samples = 1000 (Default)
num_warmup = 1000 (Default)
save_warmup = 0 (Default)
thin = 1 (Default)
adapt
engaged = 1 (Default)
gamma = 0.050000000000000003 (Default)
delta = 0.80000000000000004 (Default)
kappa = 0.75 (Default)
t0 = 10 (Default)
init_buffer = 75 (Default)
term_buffer = 50 (Default)
window = 25 (Default)
algorithm = hmc (Default)
hmc
engine = nuts (Default)
nuts
max_depth = 10 (Default)
metric = diag_e (Default)
metric_file =  (Default)
stepsize = 1 (Default)
stepsize_jitter = 0 (Default)
num_chains = 1 (Default)
id = 1 (Default)
data
file = users-guide/examples/bernoulli.data.json
init = 2 (Default)
random
seed = 68594
output
file = /tmp/tmpwxwxvhm2/bernoullirgl40flg/bernoulli-20220825133652_1.csv
diagnostic_file =  (Default)
```

(continues on next page)

(continued from previous page)

```
refresh = 100 (Default)
sig_figs = -1 (Default)
profile_file = profile.csv (Default)
num_threads = 1 (Default)


Gradient evaluation took 3e-06 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.03 seconds.
Adjust your expectations accordingly!


Iteration:    1 / 2000 [  0%]  (Warmup)
Iteration:  100 / 2000 [  5%]  (Warmup)
Iteration:  200 / 2000 [ 10%]  (Warmup)
Iteration:  300 / 2000 [ 15%]  (Warmup)
Iteration:  400 / 2000 [ 20%]  (Warmup)
Iteration:  500 / 2000 [ 25%]  (Warmup)
Iteration:  600 / 2000 [ 30%]  (Warmup)
Iteration:  700 / 2000 [ 35%]  (Warmup)
Iteration:  800 / 2000 [ 40%]  (Warmup)
Iteration:  900 / 2000 [ 45%]  (Warmup)
Iteration: 1000 / 2000 [ 50%]  (Warmup)
Iteration: 1001 / 2000 [ 50%]  (Sampling)
Iteration: 1100 / 2000 [ 55%]  (Sampling)
Iteration: 1200 / 2000 [ 60%]  (Sampling)
Iteration: 1300 / 2000 [ 65%]  (Sampling)
Iteration: 1400 / 2000 [ 70%]  (Sampling)
Iteration: 1500 / 2000 [ 75%]  (Sampling)
Iteration: 1600 / 2000 [ 80%]  (Sampling)
Iteration: 1700 / 2000 [ 85%]  (Sampling)
Iteration: 1800 / 2000 [ 90%]  (Sampling)
Iteration: 1900 / 2000 [ 95%]  (Sampling)
Iteration: 2000 / 2000 [100%]  (Sampling)

Elapsed Time: 0.008 seconds (Warm-up)
0.021 seconds (Sampling)
0.029 seconds (Total)
```

2.5 CmdStanPy Examples

2.5.1 MCMC Sampling

2.5.1.1 Overview

Stan's MCMC sampler implements the Hamiltonian Monte Carlo (HMC) algorithm and its adaptive variant the no-U-turn sampler (NUTS). It creates a set of draws from the posterior distribution of the model conditioned on the data, allowing for exact Bayesian inference of the model parameters. Each draw consists of the values for all parameter, transformed parameter, and generated quantities variables, reported on the constrained scale.

The CmdStanModel sample method wraps the CmdStan sample method. Underlyingly, the CmdStan outputs are a set of per-chain Stan CSV files. In addition to the resulting sample, reported as one row per draw, the Stan CSV files

encode information about the inference engine configuration and the sampler state. The NUTS-HMC adaptive sampler algorithm also outputs the per-chain HMC tuning parameters `step_size` and `metric`.

The `sample` method returns a CmdStanMCMC object, which provides access to the disparate information from the Stan CSV files. Accessor functions allow the user to access the sample in whatever data format is needed for further analysis, either as tabular data (i.e., in terms of the per-chain CSV file rows and columns), or as structured objects which correspond to the variables in the Stan model and the individual diagnostics produced by the inference method.

- The stan_variable and stan_variables methods return a Python numpy.ndarray containing all draws from the sample where the structure of each draw corresponds to the structure of the Stan variable.
- The draws method returns the sample as either a 2-D or 3-D numpy.ndarray.
- The draws_pd method returns the entire sample or selected variables as a pandas.DataFrame.
- The draws_xr method returns a structured Xarray dataset over the Stan model variables.
- The method_variables returns a Python dict over all sampler method variables.

In addition, the `CmdStanMCMC` object has accessor methods for

- The per-chain HMC tuning parameters `step_size` and `metric`
- The CmdStan run configuration and console outputs
- The mapping between the Stan model variables and the corresponding CSV file columns.

Notebook prerequisites

CmdStanPy displays progress bars during sampling via use of package tqdm. In order for these to display properly in a Jupyter notebook, 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.

2.5.1.2 Fitting the model and 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

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

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='bernoulli.data.json')
```

```
14:29:56 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

14:29:56 - cmdstanpy - INFO - CmdStan done processing.

```

The CmdStanMCMC object records the command, the return code, and the paths to the sampler output csv and console files. The sample is lazily instantiated on first access of either the draws or the HMC tuning parameters, i.e., the step size and metric.

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 YYYYMMDDhhmmss and the chain id, plus the corresponding filetype suffix, either '.csv' for the CmdStan output or '.txt' for the console messages, e.g. bernoulli-20220617170100_1.csv.

[4]:
```
fit
```

[4]:
```
CmdStanMCMC: model=bernoulli chains=4['method=sample', 'algorithm=hmc', 'adapt',
↪'engaged=1']
 csv_files:
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_1.csv
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_2.csv
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_3.csv
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_4.csv
 output_files:
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_0-stdout.txt
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_1-stdout.txt
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_2-stdout.txt
        /tmp/tmp576tsk6g/bernoullikolo7cku/bernoulli-20220822142956_3-stdout.txt
```

[5]:
```
print(f'draws as array:  {fit.draws().shape}')
print(f'draws as structured object:\n\t{fit.stan_variables().keys()}')
print(f'sampler diagnostics:\n\t{fit.method_variables().keys()}')
```

```
draws as array:  (1000, 4, 8)
draws as structured object:
        dict_keys(['theta'])
sampler diagnostics:
        dict_keys(['lp__', 'accept_stat__', 'stepsize__', 'treedepth__', 'n_leapfrog__',
↪'divergent__', 'energy__'])
```

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 output to the console - `show_console=True`

By default, CmdStanPy displays a progress bar during sampling, as seen above. Since the progress bars are only visible while the sampler is running and the bernoulli example model takes no time at all to fit, we run this model for 200K iterations, in order to see the progress bars in action.

[6]:
```
fit = model.sample(data='bernoulli.data.json', iter_warmup=100000, iter_sampling=100000,␣
↪show_progress=True)
```

```
14:29:56 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

14:29:58 - cmdstanpy - INFO - CmdStan done processing.

```

To see the CmdStan console outputs instead of progress bars, specify `show_console=True` This will stream all CmdStan messages to the terminal while the sampler is running. This option will allow you to debug a Stan program using the Stan language `print` statement.

[7]:
```
fit = model.sample(data='bernoulli.data.json', chains=2, parallel_chains=1, show_
↪console=True)
```

```
14:30:00 - cmdstanpy - INFO - Chain [1] start processing
14:30:00 - cmdstanpy - INFO - Chain [1] done processing
14:30:00 - cmdstanpy - INFO - Chain [2] start processing
14:30:00 - cmdstanpy - INFO - 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)
```

(continues on next page)

(continued from previous page)

```
Chain [1] window = 25 (Default)
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] id = 1
Chain [1] data
Chain [1] file = bernoulli.data.json
Chain [1] init = 2 (Default)
Chain [1] random
Chain [1] seed = 70257
Chain [1] output
Chain [1] file = /tmp/tmp576tsk6g/bernoullitkllicz1/bernoulli-20220822143000_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
Chain [1]
Chain [1]
Chain [1] Gradient evaluation took 2e-06 seconds
Chain [1] 1000 transitions using 10 leapfrog steps per transition would take 0.02␣
↪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)
Chain [1] Iteration: 1900 / 2000 [ 95%]  (Sampling)
```

(continues on next page)

(continued from previous page)

```
Chain [1] Iteration: 2000 / 2000 [100%]  (Sampling)
Chain [1]
Chain [1] Elapsed Time: 0.005 seconds (Warm-up)
Chain [1] 0.009 seconds (Sampling)
Chain [1] 0.014 seconds (Total)
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] id = 2
Chain [2] data
Chain [2] file = bernoulli.data.json
Chain [2] init = 2 (Default)
Chain [2] random
Chain [2] seed = 70257
Chain [2] output
Chain [2] file = /tmp/tmp576tsk6g/bernoullitkllicz1/bernoulli-20220822143000_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
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.
Chain [2] Adjust your expectations accordingly!
Chain [2]
Chain [2]
Chain [2] Iteration:    1 / 2000 [  0%]  (Warmup)
```

(continues on next page)

(continued from previous page)

```
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: 0.008 seconds (Warm-up)
Chain [2] 0.014 seconds (Sampling)
Chain [2] 0.022 seconds (Total)
Chain [2]
Chain [2]
```

2.5.1.3 Checking the fit

The first question to ask of the `CmdStanMCMC` object is: *is this a valid sample from the posterior?*

It is important to check whether or not the sampler was able to fit the model given the data. Often, this is not possible, for any number of reasons. To appreciate the sampler diagnostics, we use a hierarchical model which, given a small amount of data, encounters difficulty: the centered parameterization of the "8-schools" model (Rubin, 1981). The "8-schools" model is a simple hierarchical model, first developed on a dataset taken from an experiment was conducted in 8 schools, with only treatment effects and their standard errors reported.

The Stan model and the original dataset are in files `eight_schools.stan` and `eight_schools.data.json`.

eight_schools.stan

[8]:
```
with open('eight_schools.stan', 'r') as fd:
    print(fd.read())
```

```
data {
  int<lower=0> J; // number of schools
  array[J] real y; // estimated treatment effect (school j)
  array[J] real<lower=0> sigma; // std err of effect estimate (school j)
}
parameters {
  real mu;
  array[J] real theta;
```

(continues on next page)

(continued from previous page)

```
  real<lower=0> tau;
}
model {
  theta ~ normal(mu, tau);
  y ~ normal(theta, sigma);
}
```

eight_schools.data.json

[9]:
```
with open('eight_schools.data.json', 'r') as fd:
    print(fd.read())
```

```
{
    "J" : 8,
    "y" : [28,8,-3,7,-1,1,18,12],
    "sigma" : [15,10,16,11,9,11,10,18],
    "tau" : 25
}
```

Because there is not much data, the geometry of posterior distribution is highly curved, thus the sampler may encounter difficulty in fitting the model. By specifying the initial seed for the pseudo-random number generator, we insure that the sampler will have difficulty in fitting this model. In particular, some post-warmup iterations diverge, resulting in a biased sample. In addition, some post-warmup iterations hit the maximum allowed treedepth before the trajectory hits the "U-turn" condition of the NUTS algorithm, in which case the sampler may fail to properly explore the entire posterior.

These diagnostics are checked for automatically at the end of each run; if problems are detected, a WARNING message is logged.

[10]:
```
eight_schools_model = CmdStanModel(stan_file='eight_schools.stan')
eight_schools_fit = eight_schools_model.sample(data='eight_schools.data.json',␣
↪seed=55157)
```

```
14:30:00 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

14:30:00 - cmdstanpy - INFO - CmdStan done processing.
14:30:00 - cmdstanpy - WARNING - Some chains may have failed to converge.
        Chain 1 had 29 divergent transitions (2.9%)
        Chain 2 had 208 divergent transitions (20.8%)
        Chain 3 had 17 divergent transitions (1.7%)
        Chain 4 had 31 divergent transitions (3.1%)
        Use function "diagnose()" to see further information.
```

More information on how to address convergence problems can be found at https://mc-stan.org/misc/warnings

The number of post-warmup divergences and iterations which hit the maximum treedepth can be inspected directly via properties `divergences` and `max_treedepths`.

[11]:
```
print(f'divergences:\n{eight_schools_fit.divergences}\niterations at max_treedepth:\n
↪{eight_schools_fit.max_treedepths}')
```

```
divergences:
[ 29 208  17  31]
iterations at max_treedepth:
[0 0 0 0]
```

Summarizing the sample

The `summary` method reports the R-hat statistic, a measure of how well the sampler chains have converged.

[12]:
```
eight_schools_fit.summary()
```

[12]:

	Mean	MCSE	StdDev	5%	50%	95%	\
lp__	-17.80420	1.297600	5.69607	-26.306500	-18.54890	-8.18723	
mu	7.98088	0.196141	5.17030	-0.847043	8.46135	16.43540	
theta[1]	11.69530	0.357324	8.65978	-0.384251	10.32760	28.16590	
theta[2]	7.76656	0.201329	6.38375	-2.518540	7.25419	18.44390	
theta[3]	5.96852	0.239867	8.27095	-8.756270	7.21487	18.39070	
theta[4]	7.71660	0.201524	6.85139	-3.644710	8.26866	18.80780	
theta[5]	4.97621	0.447597	6.65889	-6.701880	5.63811	14.88990	
theta[6]	5.88040	0.212933	6.89437	-6.316140	6.83414	16.41370	
theta[7]	10.95250	0.243737	6.90153	0.384918	9.93285	23.58810	
theta[8]	8.47301	0.218012	8.06921	-4.222510	8.14475	22.19720	
tau	7.01515	0.772870	5.53896	1.022930	5.78582	17.32530	

	N_Eff	N_Eff/s	R_hat
lp__	19.26950	86.41040	1.18588
mu	694.85500	3115.94000	1.00830
theta[1]	587.33900	2633.81000	1.00867
theta[2]	1005.40000	4508.53000	1.00249
theta[3]	1188.96000	5331.68000	1.01065
theta[4]	1155.86000	5183.21000	1.00593
theta[5]	221.32500	992.48700	1.03708
theta[6]	1048.34000	4701.09000	1.01177
theta[7]	801.76400	3595.35000	1.00408
theta[8]	1369.94000	6143.23000	1.00222
tau	51.36157	230.32095	1.07508

Sampler Diagnostics

The `diagnose()` method provides more information about the sample.

[13]:
```
print(eight_schools_fit.diagnose())
```

```
Processing csv files: /tmp/tmp576tsk6g/eight_schoolsk2xullo4/eight_schools-
↪20220822143000_1.csv, /tmp/tmp576tsk6g/eight_schoolsk2xullo4/eight_schools-
↪20220822143000_2.csv, /tmp/tmp576tsk6g/eight_schoolsk2xullo4/eight_schools-
↪20220822143000_3.csv, /tmp/tmp576tsk6g/eight_schoolsk2xullo4/eight_schools-
↪20220822143000_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
285 of 4000 (7.12%) transitions ended with a divergence.
These divergent transitions indicate that HMC is not fully able to explore the posterior␣
↪distribution.
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, 0.28, is below the nominal threshold of 0.30 which suggests that HMC may␣
↪have trouble exploring the target distribution.
If possible, try to reparameterize the model.

Effective sample size satisfactory.

The following parameters had split R-hat greater than 1.05:
  tau
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.
```

2.5.1.4 Accessing the sampler outputs

[14]:
```
fit = model.sample(data='bernoulli.data.json')
```

```
14:30:01 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

14:30:01 - cmdstanpy - INFO - CmdStan done processing.
```

Extracting the draws as structured Stan program variables

Per-variable draws can be accessed as either a numpy.ndarray object via method `stan_variable` or as an xarray.Dataset object via `draws_xr`.

[15]:
```
print(fit.stan_variable('theta'))
```

```
[0.319739 0.384567 0.298657 ... 0.15686  0.251358 0.306462]
```

The `stan_variables` method returns a Python `dict` over all Stan variables in the output.

[16]:
```
for k, v in fit.stan_variables().items():
    print(f'name: {k}, shape: {v.shape}')
```

```
name: theta, shape: (4000,)
```

[17]:
```
print(fit.draws_xr('theta'))
```

```
<xarray.Dataset>
Dimensions:  (chain: 4, draw: 1000)
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 0.3197 0.3846 0.2987 ... 0.1569 0.2514 0.3065
Attributes:
    stan_version:        2.27.0
    model:               bernoulli_model
    num_draws_sampling:  1000
```

Extracting the draws in tabular format

The sample can be accessed either as a `numpy` array or a pandas `DataFrame`:

[18]:
```
print(f'sample as ndarray: {fit.draws().shape}\nfirst 2 draws, chain 1:\n{fit.draws()[:2,
↪ 0, :]}')
```

```
sample as ndarray: (1000, 4, 8)
first 2 draws, chain 1:
[[-6.88826   0.971817  0.998143  1.        1.        0.        6.89162
   0.319739]
 [-7.23577   0.893005  0.998143  1.        1.        0.        7.24696
   0.384567]]
```

[19]:
```
fit.draws_pd().head()
```

[19]:
```
      lp__  accept_stat__  stepsize__  treedepth__  n_leapfrog__  divergent__  \
0 -6.88826       0.971817    0.998143          1.0           1.0          0.0
1 -7.23577       0.893005    0.998143          1.0           1.0          0.0
2 -6.81820       1.000000    0.998143          2.0           3.0          0.0
```

(continues on next page)

(continued from previous page)

```
3 -6.79992       1.000000    0.998143          1.0          1.0          0.0
4 -6.80004       0.999965    0.998143          1.0          1.0          0.0

   energy__     theta
0   6.89162  0.319739
1   7.24696  0.384567
2   7.10348  0.298657
3   6.82455  0.291636
4   6.81503  0.291685
```

Extracting sampler method diagnostics

[20]:
```
for k, v in fit.method_variables().items():
    print(f'name: {k}, shape: {v.shape}')
```

```
name: lp__, shape: (1000, 4)
name: accept_stat__, shape: (1000, 4)
name: stepsize__, shape: (1000, 4)
name: treedepth__, shape: (1000, 4)
name: n_leapfrog__, shape: (1000, 4)
name: divergent__, shape: (1000, 4)
name: energy__, shape: (1000, 4)
```

Extracting the per-chain HMC tuning parameters

[21]:
```
print(f'adapted step_size per chain\n{fit.step_size}\nmetric_type: {fit.metric_type}\
↪\nmetric:\n{fit.metric}')
```

```
adapted step_size per chain
[0.998143 0.991067 0.894233 0.896342]
metric_type: diag_e
metric:
[[0.482997]
 [0.485668]
 [0.444746]
 [0.535956]]
```

Extracting the sample meta-data

[22]:
```
print('sample method variables:\n{}\n'.format(fit.metadata.method_vars_cols.keys()))
print('stan model variables:\n{}'.format(fit.metadata.stan_vars_cols.keys()))
```

```
sample method variables:
dict_keys(['lp__', 'accept_stat__', 'stepsize__', 'treedepth__', 'n_leapfrog__',
↪'divergent__', 'energy__'])

stan model variables:
dict_keys(['theta'])
```

2.5.1.5 Saving the sampler output files

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.

2.5.1.6 Parallelization via multi-threaded processing

Stan's multi-threaded processing is based on the Intel Threading Building Blocks (TBB) library, which must be linked to by the C++ compiler. To take advantage of this option, you must compile (or recompile) the program with the the C++ compiler option `STAN_THREADS`. The CmdStanModel object constructor and its `compile` method both have argument `cpp_options` which takes as its value a dictionary of compiler flags.

We compile the example model `bernoulli.stan`, this time with arguments `cpp_options` and `compile`, and use the function `exe_info()` to check that the model has been compiled for multi-threading.

[23]:
```
model = CmdStanModel(stan_file='bernoulli.stan',
                     cpp_options={'STAN_THREADS': 'TRUE'},
                     compile='force')
model.exe_info()
```

```
14:30:01 - cmdstanpy - INFO - compiling stan file /home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/bernoulli.stan to exe file /home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/bernoulli
14:30:18 - cmdstanpy - INFO - compiled model executable: /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/bernoulli
```

[23]:
```
{'stan_version_major': '2',
 'stan_version_minor': '29',
 'stan_version_patch': '0',
 'STAN_THREADS': 'true',
 'STAN_MPI': 'false',
 'STAN_OPENCL': 'false',
 'STAN_NO_RANGE_CHECKS': 'false',
 'STAN_CPP_OPTIMS': 'false'}
```

Cross-chain multi-threading

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.

The sample program argument `parallel_chains` takes an integer value which specifies how many chains to run in parallel. For models which have been compiled with option `STAN_THREADS` set, all chains are run from within a single process and the value of the `parallel_chains` argument specifies the total number of threads.

[24]:
```
fit = model.sample(data='bernoulli.data.json', parallel_chains=4)
```

```
14:30:18 - cmdstanpy - INFO - CmdStan start processing
```

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

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

14:30:18 - cmdstanpy - INFO - CmdStan done processing.
```

Within-chain multi-threading

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.

To see how this works, we run the "reflag" model, used in the reduce_sum minimal example case study. The Stan model and the original dataset are in files "redcard_reduce_sum.stan" and "redcard.json".

[25]:
```
with open('redcard_reduce_sum.stan', 'r') as fd:
    print(fd.read())
```

```
functions {
  real partial_sum(array[] int slice_n_redcards, int start, int end,
                   array[] int n_games, vector rating, vector beta) {
    return binomial_logit_lpmf(slice_n_redcards | n_games[start : end], beta[1]
                                                  + beta[2]
                                                  * rating[start :␣
↪end]);
  }
}
data {
  int<lower=0> N;
  array[N] int<lower=0> n_redcards;
  array[N] int<lower=0> n_games;
  vector[N] rating;
  int<lower=1> grainsize;
}
parameters {
  vector[2] beta;
}
model {
  beta[1] ~ normal(0, 10);
  beta[2] ~ normal(0, 1);

  target += reduce_sum(partial_sum, n_redcards, grainsize, n_games, rating,
                       beta);
}
```

As before, we compile the model specifying argument `cpp_options`.

[26]:
```
redcard_model = CmdStanModel(stan_file='redcard_reduce_sum.stan',
                     cpp_options={'STAN_THREADS': 'TRUE'},
                     compile='force')
redcard_model.exe_info()
```

```
14:30:18 - cmdstanpy - INFO - compiling stan file /home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/redcard_reduce_sum.stan to exe file /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/redcard_reduce_sum
14:30:40 - cmdstanpy - INFO - compiled model executable: /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/redcard_reduce_sum
```

[26]:
```
{'stan_version_major': '2',
 'stan_version_minor': '29',
 'stan_version_patch': '0',
 'STAN_THREADS': 'true',
 'STAN_MPI': 'false',
 'STAN_OPENCL': 'false',
 'STAN_NO_RANGE_CHECKS': 'false',
 'STAN_CPP_OPTIMS': 'false'}
```

The `sample` method argument `threads_per_chain` specifies the number of threads allotted to each chain; this corresponds to CmdStan's `num_threads` argument.

[27]:
```
redcard_fit = redcard_model.sample(data='redcard.json', threads_per_chain=4)
```

```
14:30:40 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

14:31:48 - cmdstanpy - INFO - CmdStan done processing.

```

The number of threads to use is passed to the model exe file by means of the shell environment variable `STAN_NUM_THREADS`.

On my machine, which has 4 cores, all 4 chains are run in parallel from within a single process. Therefore, the total number of threads used by this process will be `threads_per_chain` * `chains`. To check this, we examine the shell environment variable `STAN_NUM_THREADS`.

[28]:
```
os.environ['STAN_NUM_THREADS']
```

[28]:
```
'16'
```

2.5.2 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
```

```
13:36:38 - cmdstanpy - INFO - Chain [1] start processing
13:36:38 - cmdstanpy - INFO - Chain [1] done processing
```

```
('lp__', 'theta')
OrderedDict([('lp__', -5.00402), ('theta', 0.200004)])
```

[1]:

	lp__	theta
0	-5.00402	0.200004

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.

2.5.3.1 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')
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)
```

```
13:36:47 - cmdstanpy - INFO - Chain [1] start processing
13:36:47 - cmdstanpy - INFO - 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'])
```

```
0.219737
```

[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()
```

```
13:36:47 - cmdstanpy - INFO - Chain [1] start processing
13:36:47 - cmdstanpy - INFO - Chain [1] done processing
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [5], in <cell line: 2>()
      1 model_fail = CmdStanModel(stan_file='eta_should_fail.stan')
----> 2 vi_fail = model_fail.variational()
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.7/cmdstanpy/model.
↪py:1503, in CmdStanModel.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)
   1501 if len(re.findall(pat, contents)) > 0:
   1502     if require_converged:
-> 1503         raise RuntimeError(
   1504             'The algorithm may not have converged.\n'
   1505             'If you would like to inspect the output, '
   1506             're-call with require_converged=False'
   1507         )
   1508     # else:
   1509     get_logger().warning(
   1510         '%s\n%s',
   1511         'The algorithm may not have converged.',
   1512         'Proceeding because require_converged is set to False',
   1513     )

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)
```

```
13:36:48 - cmdstanpy - INFO - Chain [1] start processing
13:36:48 - cmdstanpy - INFO - Chain [1] done processing
13:36:48 - cmdstanpy - WARNING - 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]', 0.0283468),
             ('mu[2]', -0.0285787)])
```

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.

2.5.4 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.

2.5.4.1 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())
```

```
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);
}
```

2.5.4.2 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)
```

```
13:36:43 - cmdstanpy - INFO - Chain [1] start processing
13:36:43 - cmdstanpy - INFO - Chain [1] done processing
13:36:43 - cmdstanpy - WARNING - 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
)
```

```
13:36:43 - cmdstanpy - INFO - CmdStan start processing
```

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

```
chain 2 |          | 00:00 Status
```

```
chain 3 |          | 00:00 Status
```

```
chain 4 |          | 00:00 Status
```

```
13:36:44 - cmdstanpy - INFO - CmdStan done processing.
13:36:44 - cmdstanpy - WARNING - Non-fatal error during sampling:
Exception: normal_lpdf: Scale parameter is 0, but must be positive! (in '/home/docs/
↪checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/
↪examples/blr.stan', line 16, column 2 to column 45)
Exception: normal_lpdf: Scale parameter is 0, but must be positive! (in '/home/docs/
↪checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/
↪examples/blr.stan', line 16, column 2 to column 45)
Exception: normal_lpdf: Scale parameter is 0, but must be positive! (in '/home/docs/
↪checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/
↪examples/blr.stan', line 16, column 2 to column 45)
```

(continues on next page)

(continued from previous page)

```
Exception: normal_lpdf: Scale parameter is 0, but must be positive! (in '/home/docs/
↪checkouts/readthedocs.org/user_builds/cmdstanpy/checkouts/v1.0.7/docsrc/users-guide/
↪examples/blr.stan', line 16, column 2 to column 45)
Consider re-running with show_console=True if the above output is unclear!
```

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

[5]:
```
               Mean      MCSE    StdDev          5%         50%         95%  \
lp__    -156.867000  0.056908  1.743290 -160.145000 -156.532000 -154.64600
beta[1]    0.999474  0.000014  0.000952    0.997903    0.999461    1.00107
beta[2]    1.000230  0.000018  0.001160    0.998355    1.000210    1.00210
beta[3]    1.000440  0.000014  0.000938    0.998907    1.000430    1.00200
beta[4]    1.001160  0.000016  0.001064    0.999422    1.001150    1.00292
beta[5]    1.001540  0.000015  0.001033    0.999865    1.001540    1.00321
sigma      0.963840  0.004465  0.071505    0.849600    0.961783    1.09259

              N_Eff      N_Eff/s     R_hat
lp__      938.42000   987.811000  1.001480
beta[1]  4863.49000  5119.460000  1.000320
beta[2]  4344.92000  4573.600000  0.999725
beta[3]  4385.54000  4616.360000  0.999669
beta[4]  4664.71000  4910.220000  0.999536
beta[5]  4786.98000  5038.930000  0.999197
sigma     256.47117   269.969653  1.011019
```

The sampler estimates match the reference posterior.

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

```
Processing csv files: /tmp/tmpzsp0laby/blrurdrp8d3/blr-20220825133643_1.csv, /tmp/
↪tmpzsp0laby/blrurdrp8d3/blr-20220825133643_2.csv, /tmp/tmpzsp0laby/blrurdrp8d3/blr-
↪20220825133643_3.csv, /tmp/tmpzsp0laby/blrurdrp8d3/blr-20220825133643_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.
```

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)
```

```
13:36:44 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

13:36:45 - cmdstanpy - INFO - CmdStan done processing.
13:36:45 - cmdstanpy - WARNING - Some chains may have failed to converge.
        Chain 1 had 161 divergent transitions (16.1%)
        Chain 3 had 147 divergent transitions (14.7%)
        Chain 4 had 244 divergent transitions (24.4%)
        Use function "diagnose()" to see further information.
```

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

[8]:

	Mean	MCSE	StdDev	5%	50%	95%	\
lp__	-191.333000	24.678800	35.170300	-231.560000	-165.541000	-154.33900	
beta[1]	0.999452	0.000119	0.001816	0.996272	0.999494	1.00252	
beta[2]	1.000560	0.000229	0.002416	0.996529	1.000410	1.00459	
beta[3]	1.000590	0.000259	0.002043	0.997326	1.000480	1.00442	
beta[4]	1.001380	0.000224	0.002279	0.997013	1.001690	1.00512	
beta[5]	1.001200	0.000150	0.002013	0.997854	1.001290	1.00443	
sigma	1.962000	0.725020	1.034300	0.907470	2.708830	3.17346	

	N_Eff	N_Eff/s	R_hat
lp__	2.03097	6.28785	11.37150
beta[1]	232.18100	718.82600	1.01286
beta[2]	110.88200	343.28900	1.04571
beta[3]	62.47110	193.40900	1.04607
beta[4]	103.34500	319.95200	1.09049
beta[5]	180.70500	559.45900	1.03165
sigma	2.03514	6.30075	10.50420

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

```
Processing csv files: /tmp/tmpzsp0laby/blrittrbo3n/blr-20220825133644_1.csv, /tmp/
↪tmpzsp0laby/blrittrbo3n/blr-20220825133644_2.csv, /tmp/tmpzsp0laby/blrittrbo3n/blr-
↪20220825133644_3.csv, /tmp/tmpzsp0laby/blrittrbo3n/blr-20220825133644_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
552 of 4000 (13.80%) transitions ended with a divergence.
These divergent transitions indicate that HMC is not fully able to explore the posterior␣
↪distribution.
Try increasing adapt delta closer to 1.
```

(continues on next page)

(continued from previous page)

```
If this doesn't remove all divergences, try to reparameterize the model.

Checking E-BFMI - sampler transitions HMC potential energy.
The E-BFMI, 0.01, is below the nominal threshold of 0.30 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 1.05:
  beta[4], 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.
```

2.5.5 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

2.5.5.1 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')
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)
print(model.code())
```

```
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)
```

```
13:36:40 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status
chain 2 |          | 00:00 Status
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status

13:36:40 - cmdstanpy - INFO - 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  \
lp__   -7.295440  0.021688  0.748366 -8.916780 -6.999430 -6.750240  1190.69
```

(continues on next page)

(continued from previous page)

```
theta  0.246662  0.003183  0.119731  0.069531  0.235259  0.459562  1415.35

       N_Eff/s    R_hat
lp__   15072.0  1.00102
theta  17915.8  1.00274
```

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())
```

```
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);
  y ~ bernoulli(theta);
}
generated quantities {
  array[N] int y_rep;
  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)
```

```
13:36:40 - cmdstanpy - INFO - Chain [1] start processing
13:36:40 - cmdstanpy - INFO - Chain [1] done processing
13:36:40 - cmdstanpy - INFO - Chain [2] start processing
13:36:40 - cmdstanpy - INFO - Chain [2] done processing
13:36:40 - cmdstanpy - INFO - Chain [3] start processing
13:36:40 - cmdstanpy - INFO - Chain [3] done processing
13:36:40 - cmdstanpy - INFO - Chain [4] start processing
13:36:40 - cmdstanpy - INFO - 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. 0. 0. 0. 1. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
[[0. 0. 0. 0. 1. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 1. 1. 1. 0. 1. 1.]
 [0. 1. 0. 0. 1. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 1. 0. 0. 0.]]
[[1. 0. 0. 1. 0. 0. 1. 0. 1. 1.]
 [1. 0. 0. 0. 0. 0. 1. 0. 1. 1.]
 [1. 0. 0. 0. 0. 0. 1. 0. 1. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

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	-6.99134	0.879873	0.730324	1.0	3.0	0.0	7.69176	0.343125	0.0	0.0	
1	-6.93214	0.869749	0.730324	2.0	3.0	0.0	7.98512	0.179704	0.0	0.0	
2	-6.85734	0.959579	0.730324	1.0	3.0	0.0	7.32725	0.311241	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	1.0	0.0	1.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	1.0	0.0	1.0	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.

2.5.6 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()
```

```
14:52:06 - cmdstanpy - INFO - compiling stan file /home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/bernoulli_external.stan to exe file /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/bernoulli_external
14:52:06 - cmdstanpy - ERROR - Stan program failed to compile:
14:52:06 - cmdstanpy - WARNING -
--- Translating Stan model to C++ code ---
bin/stanc  --o=/home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/bernoulli_
↪external.hpp /home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/bernoulli_
↪external.stan
Semantic error in '/home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/bernoulli_
↪external.stan', line 2, column 7 to column 16:
   -------------------------------------------------
     1:  functions {
     2:    real make_odds(real theta);
                ^
     3:  }
     4:  data {
   -------------------------------------------------

Function is declared without specifying a definition.
```

(continues on next page)

(continued from previous page)

```
make: *** [make/program:50: /home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/
↪bernoulli_external.hpp] Error 1

Command ['make', '/home/brian/Dev/py/cmdstanpy/docsrc/users-guide/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})
```

```
14:52:06 - cmdstanpy - INFO - compiling stan file /home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/bernoulli_external.stan to exe file /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/bernoulli_external
14:52:07 - cmdstanpy - ERROR - Stan program failed to compile:
14:52:07 - cmdstanpy - WARNING -
--- Translating Stan model to C++ code ---
bin/stanc --allow-undefined --o=/home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/
↪bernoulli_external.hpp /home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/
↪bernoulli_external.stan

--- Compiling, linking C++ code ---
g++ -march=native -mtune=native -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-
↪ignored-attributes      -I stan/lib/stan_math/lib/tbb_2020.3/include    -O3 -I src -I␣
↪stan/src -I lib/rapidjson_1.1.0/ -I lib/CLI11-1.9.1/ -I stan/lib/stan_math/ -I stan/
↪lib/stan_math/lib/eigen_3.3.9 -I stan/lib/stan_math/lib/boost_1.75.0 -I stan/lib/stan_
↪math/lib/sundials_6.0.0/include -I stan/lib/stan_math/lib/sundials_6.0.0/src/sundials ␣
↪  -DBOOST_DISABLE_ASSERTS          -c -Wno-ignored-attributes   -include /home/brian/
↪Dev/py/cmdstanpy/docsrc/users-guide/examples/user_header.hpp -x c++ -o /home/brian/Dev/
↪py/cmdstanpy/docsrc/users-guide/examples/bernoulli_external.o /home/brian/Dev/py/
↪cmdstanpy/docsrc/users-guide/examples/bernoulli_external.hpp
cc1plus: fatal error: /home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/user_
↪header.hpp: No such file or directory
compilation terminated.
make: *** [make/program:58: /home/brian/Dev/py/cmdstanpy/docsrc/users-guide/examples/
↪bernoulli_external] Error 1

Command ['make', 'STANCFLAGS+=--allow-undefined', '/home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/bernoulli_external']
        error during processing No such file or directory
```

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')
assert model_external.exe_file is not None
```

```
14:52:07 - cmdstanpy - INFO - compiling stan file /home/brian/Dev/py/cmdstanpy/docsrc/
↪users-guide/examples/bernoulli_external.stan to exe file /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/bernoulli_external
14:52:25 - cmdstanpy - INFO - compiled model executable: /home/brian/Dev/py/cmdstanpy/
↪docsrc/users-guide/examples/bernoulli_external
```

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')
```

```
14:52:25 - cmdstanpy - INFO - CmdStan start processing
chain 1 |          | 00:00 Status


chain 1 || 00:00 Sampling completed
chain 2 || 00:00 Sampling completed
chain 3 || 00:00 Sampling completed
chain 4 || 00:00 Sampling completed


14:52:25 - cmdstanpy - INFO - CmdStan done processing.
```

[6]:
```
array([0.20418  , 0.27078  , 0.614841 , ..., 0.124882 , 0.0797855,
       0.165672 ])
```

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 {
    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

THREE

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.

3.1 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*.

3.1.1 Classes

3.1.1.1 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.

property stan_vars_types: Dict[str, cmdstanpy.utils.stancsv.BaseType]

Returns map from Stan program variable names to variable base type. Uses deepcopy for immutability.

3.1.1.2 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.

3.1.1.3 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[Union[str, os.PathLike]]*) –

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.

3.1.1.4 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[Union[str, os.PathLike]]*) –
- **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[Union[str, os.PathLike]]*) –
- **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

3.1.1.5 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

3.1.1.6 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

3.1.1.7 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

3.2 Classes

3.2.1 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[Union[str, os.PathLike]]*) – Path to Stan program file.
- **exe_file** (*Optional[Union[str, os.PathLike]]*) – 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[Union[str, os.PathLike]]*) – 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[Union[str, os.PathLike]]*) – 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]

format(*overwrite_file=False*, *canonicalize=False*, *max_line_length=78*, ***, *backup=True*)

Run stanc's auto-formatter on the model code. Either saves directly back to the file or prints for inspection

Parameters

- **overwrite_file** (*bool*) – If True, save the updated code to disk, rather than printing it. By default False
- **canonicalize** (*Union[bool, str, Iterable[str]]*) – Whether or not the compiler should 'canonicalize' the Stan model, removing things like deprecated syntax. Default is False. If True, all canonicalizations are run. If it is a list of strings, those options are passed to stanc (new in Stan 2.29)
- **max_line_length** (*int*) – Set the wrapping point for the formatter. The default value is 78, which wraps most lines by the 80th character.
- **backup** (*bool*) – If True, create a stanfile.bak backup before writing to the file. Only disable this if you're sure you have other copies of the file or are using a version control system like Git.

Return type None

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, os.PathLike]]*) – 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.mcmc.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[Union[str, os.PathLike]]*) – 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.gq.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, os.PathLike]]*) – 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, os.PathLike]]*) – 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[Union[str, os.PathLike]]*) – 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.mle.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, os.PathLike]]*) – 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[Union[str, os.PathLike]]*) – 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.mcmc.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, os.PathLike]]*) – 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[Union[str, os.PathLike]]*) – 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.vb.CmdStanVB*

property cpp_options: Dict[str, Union[bool, int]]

Options to C++ compilers.

property exe_file: Optional[Union[str, os.PathLike]]

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[Union[str, os.PathLike]]

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

3.2.2 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.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*, *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).

This functionaltiy is also available via a shortcut using `.` - writing `fit.a` is a synonym for `fit.stan_variable("a")`

Parameters

- **var** (*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=(5, 50, 95)*, *sig_figs=6*)

Run cmdstan/bin/stansummary over all output CSV files, assemble summary into DataFrame object. The first row contains statistics for the total joint log probability *lp__*, but is omitted when the Stan model has no parameters. The remaining rows contain summary statistics for all parameters, transformed parameters, and generated quantities variables, in program declaration order.

Parameters

- **percentiles** (*Sequence[int]*) – Ordered non-empty sequence of percentiles to report. Must be integers from (1, 99), inclusive. Defaults to `(5, 50, 95)`
- **sig_figs** (*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 divergences: Optional[numpy.ndarray]

Per-chain total number of post-warmup divergent iterations. When sampler algorithm 'fixed_param' is specified, returns None.

property max_treedepths: Optional[numpy.ndarray]

Per-chain total number of post-warmup iterations where the NUTS sampler reached the maximum allowed treedepth. When sampler algorithm 'fixed_param' is specified, returns None.

property metadata: *cmdstanpy.stanfit.metadata.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', according to CmdStan arg 'metric'. 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).

3.2.3 CmdStanMLE

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

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

Parameters **runset** (cmdstanpy.stanfit.runset.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*, ***, *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.

This functionaltiy is also available via a shortcut using `.` - writing `fit.a` is a synonym for `fit.stan_variable("a")`

Parameters

- **var** (*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.metadata.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.

3.2.4 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.RunSet) –
- **mcmc_sample** (cmdstanpy.stanfit.mcmc.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*, *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).

This functionaltiy is also available via a shortcut using `.` - writing `fit.a` is a synonym for `fit.stan_variable("a")`

Parameters

- **var** (*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.metadata.InferenceMetadata*

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

3.2.5 CmdStanVB

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

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

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

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.

This functionaltiy is also available via a shortcut using `.` - writing `fit.a` is a synonym for `fit.stan_variable("a")`

Parameters **var** (*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.metadata.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.

3.3 Functions

3.3.1 show_versions

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

Prints out system and dependency information for debugging

Parameters **output** (*bool*) –

Return type str

3.3.2 cmdstan_path

cmdstanpy.**cmdstan_path**()

Validate, then return CmdStan directory path.

Return type str

3.3.3 install_cmdstan

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

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. "2.29.2". 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.
- **interactive** (*bool*) – Boolean value; if true, ignore all other arguments to this function and run in an interactive mode, prompting the user to provide the other information manually through the standard input.

  This flag should only be used in interactive environments, e.g. on the command line.

Returns Boolean value; `True` for success.

Return type bool

3.3.4 set_cmdstan_path

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

Validate, then set CmdStan directory path.

Parameters **path** (*str*) –

Return type None

3.3.5 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, ...]]

3.3.6 set_make_env

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

set MAKE environmental variable.

Parameters **make** (*str*) –

Return type None

3.3.7 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[str, List[str], os.PathLike]]*) – directory path
- **method** (*Optional[str]*) – method name (optional)

Returns either a CmdStanMCMC, CmdStanMLE, or CmdStanVB object

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

3.3.8 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

CHAPTER

FOUR

WHAT'S NEW

For full changes, see the Releases page on GitHub.

4.1 CmdStanPy 1.0.6

- Fixed an issue where complex number containers in Stan program outputs were not being read in properly by CmdStanPy. The output would have the correct shape, but the values would be mixed up.

4.2 CmdStanPy 1.0.6

- Fixed a build error in the documentation
- Improved messages when model fails to compile due to C++ errors.

4.3 CmdStanPy 1.0.5

- Fixed a typo in *cmdstanpy.show_versions()*
- Reorganized and updated the documentation
- Reorganized a lot of internal code
- Cleaned up the output of *CmdStanMCMC.draws_pd()*
- Cleaned up the output of *CmdStanMCMC.summary()*
- Removed the logging which occurred when Python exited with cmdstanpy imported.

4.4 CmdStanPy 1.0.4

- Fix an issue with *cmdstanpy.install_cmdstan()* where the installation would report that it had failed even when it had not.

4.5 CmdStanPy 1.0.3

- Fix an issue where Stan fit objects were not `pickle`-able when they previously were.

> **Warning:** We still do not recommend pickling cmdstanpy objects, but rather using functions *save_csvfiles()* and *from_csv()*.

4.6 CmdStanPy 1.0.2

- CmdStanPy can now format (and canonicalize) your Stan files with *CmdStanModel.format()*
- Stan variables can now be accessed from fit objects using the . syntax when no naming conflicts occur. For example, previous code `fit.stan_variable("my_cool_variable")` can now be written `fit.my_cool_variable`
- CmdStanPy is more robust to running in threaded environments and tries harder to not overwrite its own output files
- The `install_cmdstan` script can now be run in interactive mode using `--interactive/-i`
- CmdStanPy now computes some diagnostics after running HMC and will warn you about post-warmup divergences and treedepth exceptions
- Runtime exceptions in the `generated quantities` block should be recognized better now.
- The default level of precision used by *CmdStanMCMC.summary()* is now 6, as it is when `stansummary` is used from the command line.
- Various documentation improvements

4.7 CmdStanPy 1.0.1

- Support new optimizations in CmdStan 2.29
- Support complex numbers as both inputs and outputs of Stan programs
- Sped up assembling output by only reading draws at most once
- Fixed an issue where a command failing could change your working directory
- Improve error messages in some cases
- CmdStanPy no longer changes the global root logging level

Note: The minimum supported version for CmdStanPy is now Python 3.7.

4.8 CmdStanPy 1.0.0

- Initial release

CHAPTER

FIVE

COMMUNITY

This page highlights community projects that involve CmdStanPy. Check them out!

5.1 Project templates

Templates are a great way to piggy back on other users' work, saving you time when you start a new project.

- cookiecutter-cmdstanpy-analysis A cookiecutter template for cmdstanpy-based statistical analysis projects.
- cookiecutter-cmdstanpy-wrapper A cookiecutter template using Stan models in Python packages, including the ability to pre-compile the model as part of the package distribution.

5.2 Software

- Prophet A procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects.
- ArviZ A Python package (with a Julia interface) for exploratory analysis of Bayesian models. Includes functions for posterior analysis, data storage, model checking, comparison and diagnostics.

genindex

PYTHON MODULE INDEX

C

cmdstanpy, ??

INDEX

A

add() (*cmdstanpy.compiler_opts.CompilerOptions method*), 51
add_include_path() (*cmdstanpy.compiler_opts.CompilerOptions method*), 51

C

chain_ids (*cmdstanpy.CmdStanGQ property*), 72
chain_ids (*cmdstanpy.CmdStanMCMC property*), 68
chain_ids (*cmdstanpy.stanfit.RunSet property*), 50
chains (*cmdstanpy.CmdStanGQ property*), 72
chains (*cmdstanpy.CmdStanMCMC property*), 68
chains (*cmdstanpy.stanfit.RunSet property*), 50
cmd() (*cmdstanpy.stanfit.RunSet method*), 50
cmdstan_config (*cmdstanpy.InferenceMetadata property*), 49
cmdstan_path() (*in module cmdstanpy*), 74
cmdstan_version() (*in module cmdstanpy*), 75
CmdStanArgs (*class in cmdstanpy.cmdstan_args*), 52
CmdStanGQ (*class in cmdstanpy*), 70
CmdStanMCMC (*class in cmdstanpy*), 65
CmdStanMLE (*class in cmdstanpy*), 69
CmdStanModel (*class in cmdstanpy*), 56
cmdstanpy
    module, 1
CmdStanVB (*class in cmdstanpy*), 73
code() (*cmdstanpy.CmdStanModel method*), 57
column_names (*cmdstanpy.CmdStanGQ property*), 72
column_names (*cmdstanpy.CmdStanMCMC property*), 68
column_names (*cmdstanpy.CmdStanMLE property*), 69
column_names (*cmdstanpy.CmdStanVB property*), 73
columns (*cmdstanpy.CmdStanVB property*), 73
compile() (*cmdstanpy.CmdStanModel method*), 57
CompilerOptions (*class in cmdstanpy.compiler_opts*), 51
compose() (*cmdstanpy.cmdstan_args.OptimizeArgs method*), 55
compose() (*cmdstanpy.cmdstan_args.SamplerArgs method*), 54
compose() (*cmdstanpy.cmdstan_args.VariationalArgs method*), 55
compose() (*cmdstanpy.compiler_opts.CompilerOptions method*), 51
compose_command() (*cmdstanpy.cmdstan_args.CmdStanArgs method*), 53
cpp_options (*cmdstanpy.CmdStanModel property*), 64
cpp_options (*cmdstanpy.compiler_opts.CompilerOptions property*), 52
csv_files (*cmdstanpy.stanfit.RunSet property*), 50

D

diagnose() (*cmdstanpy.CmdStanMCMC method*), 65
diagnostic_files (*cmdstanpy.stanfit.RunSet property*), 50
divergences (*cmdstanpy.CmdStanMCMC property*), 68
draws() (*cmdstanpy.CmdStanGQ method*), 70
draws() (*cmdstanpy.CmdStanMCMC method*), 65
draws_pd() (*cmdstanpy.CmdStanGQ method*), 71
draws_pd() (*cmdstanpy.CmdStanMCMC method*), 66
draws_xr() (*cmdstanpy.CmdStanGQ method*), 71
draws_xr() (*cmdstanpy.CmdStanMCMC method*), 66

E

eta (*cmdstanpy.CmdStanVB property*), 73
exe_file (*cmdstanpy.CmdStanModel property*), 65
exe_info() (*cmdstanpy.CmdStanModel method*), 57

F

format() (*cmdstanpy.CmdStanModel method*), 57
from_csv() (*in module cmdstanpy*), 76

G

generate_quantities() (*cmdstanpy.CmdStanModel method*), 57
get_err_msgs() (*cmdstanpy.stanfit.RunSet method*), 50

I

InferenceMetadata (*class in cmdstanpy*), 49
install_cmdstan() (*in module cmdstanpy*), 74

is_empty() (*cmdstanpy.compiler_opts.CompilerOptions method*), 52

M

max_treedepths (*cmdstanpy.CmdStanMCMC property*), 68
metadata (*cmdstanpy.CmdStanGQ property*), 72
metadata (*cmdstanpy.CmdStanMCMC property*), 68
metadata (*cmdstanpy.CmdStanMLE property*), 69
metadata (*cmdstanpy.CmdStanVB property*), 73
method (*cmdstanpy.stanfit.RunSet property*), 50
method_variables() (*cmdstanpy.CmdStanMCMC method*), 66
method_vars_cols (*cmdstanpy.InferenceMetadata property*), 49
metric (*cmdstanpy.CmdStanMCMC property*), 68
metric_type (*cmdstanpy.CmdStanMCMC property*), 68
model (*cmdstanpy.stanfit.RunSet property*), 51
module
    cmdstanpy, 1

N

name (*cmdstanpy.CmdStanModel property*), 65
num_draws_sampling (*cmdstanpy.CmdStanMCMC property*), 68
num_draws_warmup (*cmdstanpy.CmdStanMCMC property*), 68
num_procs (*cmdstanpy.stanfit.RunSet property*), 51

O

one_process_per_chain (*cmdstanpy.stanfit.RunSet property*), 51
optimize() (*cmdstanpy.CmdStanModel method*), 58
OptimizeArgs (*class in cmdstanpy.cmdstan_args*), 54
optimized_iterations_np (*cmdstanpy.CmdStanMLE property*), 70
optimized_iterations_pd (*cmdstanpy.CmdStanMLE property*), 70
optimized_params_dict (*cmdstanpy.CmdStanMLE property*), 70
optimized_params_np (*cmdstanpy.CmdStanMLE property*), 70
optimized_params_pd (*cmdstanpy.CmdStanMLE property*), 70

P

profile_files (*cmdstanpy.stanfit.RunSet property*), 51

R

RunSet (*class in cmdstanpy.stanfit*), 50

S

sample() (*cmdstanpy.CmdStanModel method*), 60
SamplerArgs (*class in cmdstanpy.cmdstan_args*), 53
save_csvfiles() (*cmdstanpy.CmdStanGQ method*), 71
save_csvfiles() (*cmdstanpy.CmdStanMCMC method*), 66
save_csvfiles() (*cmdstanpy.CmdStanMLE method*), 69
save_csvfiles() (*cmdstanpy.CmdStanVB method*), 73
save_csvfiles() (*cmdstanpy.stanfit.RunSet method*), 50
set_cmdstan_path() (*in module cmdstanpy*), 75
set_make_env() (*in module cmdstanpy*), 75
show_versions() (*in module cmdstanpy*), 74
src_info() (*cmdstanpy.CmdStanModel method*), 63
stan_file (*cmdstanpy.CmdStanModel property*), 65
stan_variable() (*cmdstanpy.CmdStanGQ method*), 71
stan_variable() (*cmdstanpy.CmdStanMCMC method*), 67
stan_variable() (*cmdstanpy.CmdStanMLE method*), 69
stan_variable() (*cmdstanpy.CmdStanVB method*), 73
stan_variables() (*cmdstanpy.CmdStanGQ method*), 72
stan_variables() (*cmdstanpy.CmdStanMCMC method*), 67
stan_variables() (*cmdstanpy.CmdStanMLE method*), 69
stan_variables() (*cmdstanpy.CmdStanVB method*), 73
stan_vars_cols (*cmdstanpy.InferenceMetadata property*), 49
stan_vars_dims (*cmdstanpy.InferenceMetadata property*), 49
stan_vars_types (*cmdstanpy.InferenceMetadata property*), 49
stanc_options (*cmdstanpy.CmdStanModel property*), 65
stanc_options (*cmdstanpy.compiler_opts.CompilerOptions property*), 52
stdout_files (*cmdstanpy.stanfit.RunSet property*), 51
step_size (*cmdstanpy.CmdStanMCMC property*), 68
summary() (*cmdstanpy.CmdStanMCMC method*), 67

T

thin (*cmdstanpy.CmdStanMCMC property*), 68

U

user_header (*cmdstanpy.CmdStanModel property*), 65
user_header (*cmdstanpy.compiler_opts.CompilerOptions property*), 52

V

validate() (*cmdstanpy.cmdstan_args.CmdStanArgs method*), 53

validate() (*cmdstanpy.cmdstan_args.OptimizeArgs method*), 55
validate() (*cmdstanpy.cmdstan_args.SamplerArgs method*), 54
validate() (*cmdstanpy.cmdstan_args.VariationalArgs method*), 56
validate() (*cmdstanpy.compiler_opts.CompilerOptions method*), 52
validate_cpp_opts() (*cmdstanpy.compiler_opts.CompilerOptions method*), 52
validate_stanc_opts() (*cmdstanpy.compiler_opts.CompilerOptions method*), 52
validate_user_header() (*cmdstanpy.compiler_opts.CompilerOptions method*), 52
variational() (*cmdstanpy.CmdStanModel method*), 63
variational_params_dict (*cmdstanpy.CmdStanVB property*), 74
variational_params_np (*cmdstanpy.CmdStanVB property*), 74
variational_params_pd (*cmdstanpy.CmdStanVB property*), 74
variational_sample (*cmdstanpy.CmdStanVB property*), 74
VariationalArgs (*class in cmdstanpy.cmdstan_args*), 55

W

write_stan_json() (*in module cmdstanpy*), 76