PyMC + ArviZ

Lecture 19

Dr. Colin Rundel

pymc

> PyMC is a probabilistic programming library for Python that allows users to build Bayesian models with a simple Python API and fit them using Markov chain Monte Carlo (MCMC) methods.
>
> - **Modern** - Includes state-of-the-art inference algorithms, including MCMC (NUTS) and variational inference (ADVI).
> - **User friendly** - Write your models using friendly Python syntax. Learn Bayesian modeling from the many example notebooks.
> - **Fast** - Uses Aesara as its computational backend to compile to C and JAX, run your models on the GPU, and benefit from complex graph-optimizations.
> - **Batteries included** - Includes probability distributions, Gaussian processes, ABC, SMC and much more. It integrates nicely with ArviZ for visualizations and diagnostics, as well as Bambi for high-level mixed-effect models.
> - **Community focused** - Ask questions on discourse, join MeetUp events, follow us on Twitter, and start contributing.

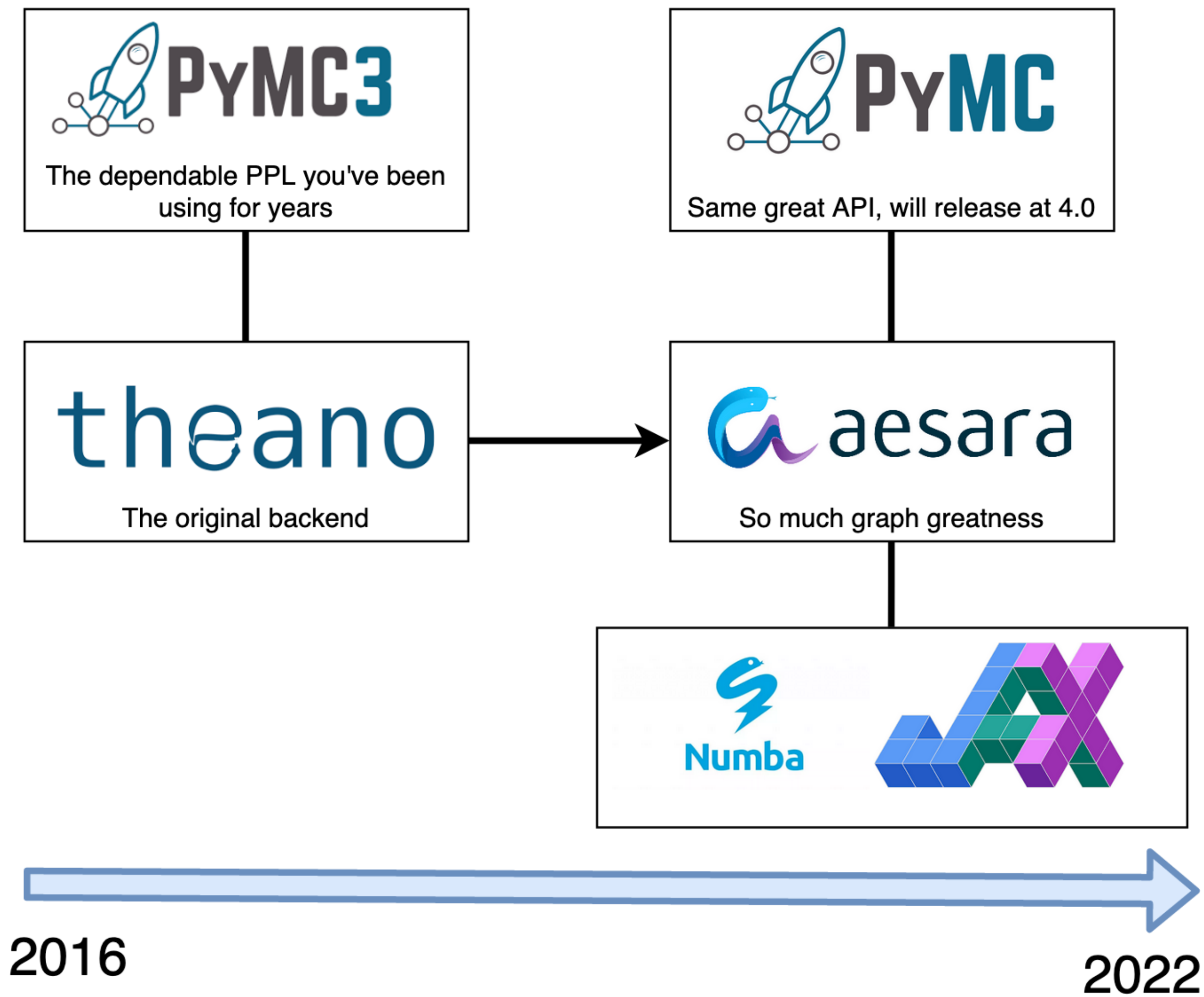
```python
import pymc as pm
```

ArviZ

> ArviZ is a Python package for exploratory analysis of Bayesian models. Includes functions for posterior analysis, data storage, sample diagnostics, model checking, and comparison.
>
> - **Interoperability** - Integrates with all major probabilistic programming libraries: PyMC, CmdStanPy, PyStan, Pyro, NumPyro, and emcee.
> - **Large Suite of Visualizations** - Provides over 25 plotting functions for all parts of Bayesian workflow: visualizing distributions, diagnostics, and model checking. See the gallery for examples.
> - **State of the Art Diagnostics** - Latest published diagnostics and statistics are implemented, tested and distributed with ArviZ.
> - **Flexible Model Comparison** - Includes functions for comparing models with information criteria, and cross validation (both approximate and brute force).
> - **Built for Collaboration** - Designed for flexible cross-language serialization using netCDF or Zarr formats. ArviZ also has a Julia version that uses the same data schema.
> - **Labeled Data** - Builds on top of xarray to work with labeled dimensions and coordinates.

```python
import arviz as az
```

Some history



Graphic by Ravin Kumar from PyMC 4.0 Release Announcement

Model basics

All models are derived from the `Model()` class, unlike what we have seen previously PyMC makes heavy use of Python's context manager using the `with` statement to add model components to a model.

```
with pm.Model() as norm:
  x = pm.Normal("x", mu=0, sigma=1)
```

```
x = pm.Normal("x", mu=0, sigma=1)
```

```
Error: TypeError: No model on context stack, which is needed to instantiate distributions. Add variable insi
```

Note that `with` blocks do not have their own scope - so variables defined inside are added to the parent scope (becareful about overwriting other variables).

```
x
```

```
x
```

```
type(x)
```

```
<class 'pytensor.tensor.var.TensorVariable'>
```

Random Variables

`pm.Normal()` is an example of a PyMC distribution, which are used to construct models, these are implemented using the `TensorVariable` class which is used for all of the builtin distributions (and can be used to create custom distributions). Generally you will not be interacting with these objects directly, but with that said some useful methods and attributes:

```
type(norm.x)
```

```
<class 'pytensor.tensor.var.TensorVariable'>
```

```
norm.x.owner.op
```

```
<pytensor.tensor.random.basic.NormalRV object at 0x17280e980>
```

```
pm.draw(norm.x)
```

```
array(0.16579)
```

Standalone RVs

If you really want to construct a `TensorVariable` outside of a model this can be done via the `dist` method for each distribution.

```
z = pm.Normal.dist(mu=1, sigma=2, shape=[2,3])
z
```

```
normal_rv{0, (0, 0), floatX, False}.out
```

```
pm.draw(z)
```

```
array([[ 1.83488,  4.45201, -1.60857],
       [-0.29248,  1.7887 ,  1.81323]])
```

Modifying models

Because of this construction it is possible to add additional components to an existing (named) model via subsequent `with` statements (only the first needs `pm.Model()`)

```
with norm:
  y = pm.Normal("y", mu=x, sigma=1, shape=3)
```

```
norm.basic_RVs
```

```
[x, y]
```

Variable heirarchy

Note that we defined $y|x \sim \quad (x, 1)$, so what is happening when we use `pm.draw(norm.y)`?

```
pm.draw(norm.y)
```

```
array([-1.49714, -1.75064, -0.81631])
```

```
obs = pm.draw(norm.y, draws=1000); obs
```

```
array([[ 2.2078 ,  2.29494,  1.4284 ],
       [-0.93994, -0.55348,  0.15047],
       [ 0.54024,  1.27989,  0.67047],
       ...,
       [ 1.6475 ,  0.83945,  0.71665],
       [-0.18381, -0.30989, -0.36771],
       [-0.52248, -0.31888, -0.42917]])
```

```
np.mean(obs)
```

```
0.06400085943586256
```

```
np.var(obs)
```

```
1.9553522392586231
```

```
np.std(obs)
```

```
1.3983391002395031
```

Each time we ask for a draw from `y`, PyMC is first drawing from `x` for us.

Beta-Binomial model

We will now build a basic model where we know what the solution should look like and compare the results.

```
with pm.Model() as beta_binom:
  p = pm.Beta("p", alpha=10, beta=10)
  x = pm.Binomial("x", n=20, p=p, observed=5)
```

In order to sample from the posterior we add a call to `sample()` within the model context.

```
with beta_binom:
  trace = pm.sample(random_seed=1234, progressbar=False)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [p]
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 0 seconds.
```

`pm.sample()` results

```python
print(trace)
```

```
Inference data with groups:
	> posterior
	> sample_stats
	> observed_data
```

```python
print(type(trace))
```

```
<class 'arviz.data.inference_data.InferenceData'>
```

Xarray - N-D labeled arrays and datasets in Python

> Xarray (formerly xray) is an open source project and Python package that makes working with labelled multi-dimensional arrays simple, efficient, and fun!
> Xarray introduces labels in the form of dimensions, coordinates and attributes on top of raw NumPy-like arrays, which allows for a more intuitive, more concise, and less error-prone developer experience. The package includes a large and growing library of domain-agnostic functions for advanced analytics and visualization with these data structures.
> Xarray is inspired by and borrows heavily from pandas, the popular data analysis package focused on labelled tabular data. It is particularly tailored to working with netCDF files, which were the source of xarray's data model, and integrates tightly with dask for parallel computing.

See here for more details on xarray

Digging into `trace`

```
print(trace.posterior)
```

```
<xarray.Dataset>
Dimensions:  (chain: 4, draw: 1000)
Coordinates:
  * chain    (chain) int64 0 1 2 3
  * draw     (draw) int64 0 1 2 3 4 5 6 7 8 ... 992 993 994 995 996 997 998 999
Data variables:
    p        (chain, draw) float64 0.5068 0.4518 0.3853 ... 0.166 0.3242 0.3242
Attributes:
    created_at:                 2023-03-24T16:28:00.773567
    arviz_version:              0.15.1
    inference_library:          pymc
    inference_library_version:  5.1.2
    sampling_time:              0.32303428649902344
    tuning_steps:               1000
```

```
print(trace.posterior["p"].shape)
```

```
(4, 1000)
```

```
print(trace.sel(chain=0).posterior["p"].shape)
```

```
(1000,)
```

```
print(trace.sel(draw=slice(500, None, 10)).posterior["p"].shape)
```

```
(4, 50)
```

As DataFrame

Posterior values, or subsets, can be converted to DataFrames via the `to_dataframe()` method

```
trace.posterior.to_dataframe()
```

```
                      p
chain draw
0     0      0.506797
      1      0.451803
      2      0.385329
      3      0.507061
      4      0.374281
...               ...
3     995    0.284547
      996    0.202096
      997    0.165955
      998    0.324233
      999    0.324233

[4000 rows x 1 columns]
```

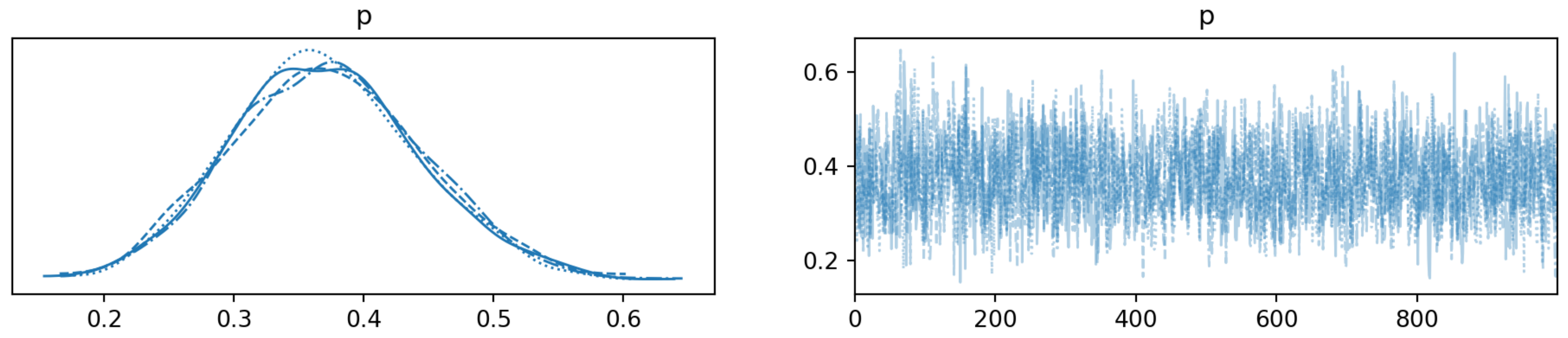
```
trace.posterior["p"][0,:].to_dataframe()
```

```
      chain         p
draw
0         0  0.506797
1         0  0.451803
2         0  0.385329
3         0  0.507061
4         0  0.374281
...     ...       ...
995       0  0.356664
996       0  0.356664
997       0  0.397380
998       0  0.401865
999       0  0.401865

[1000 rows x 2 columns]
```

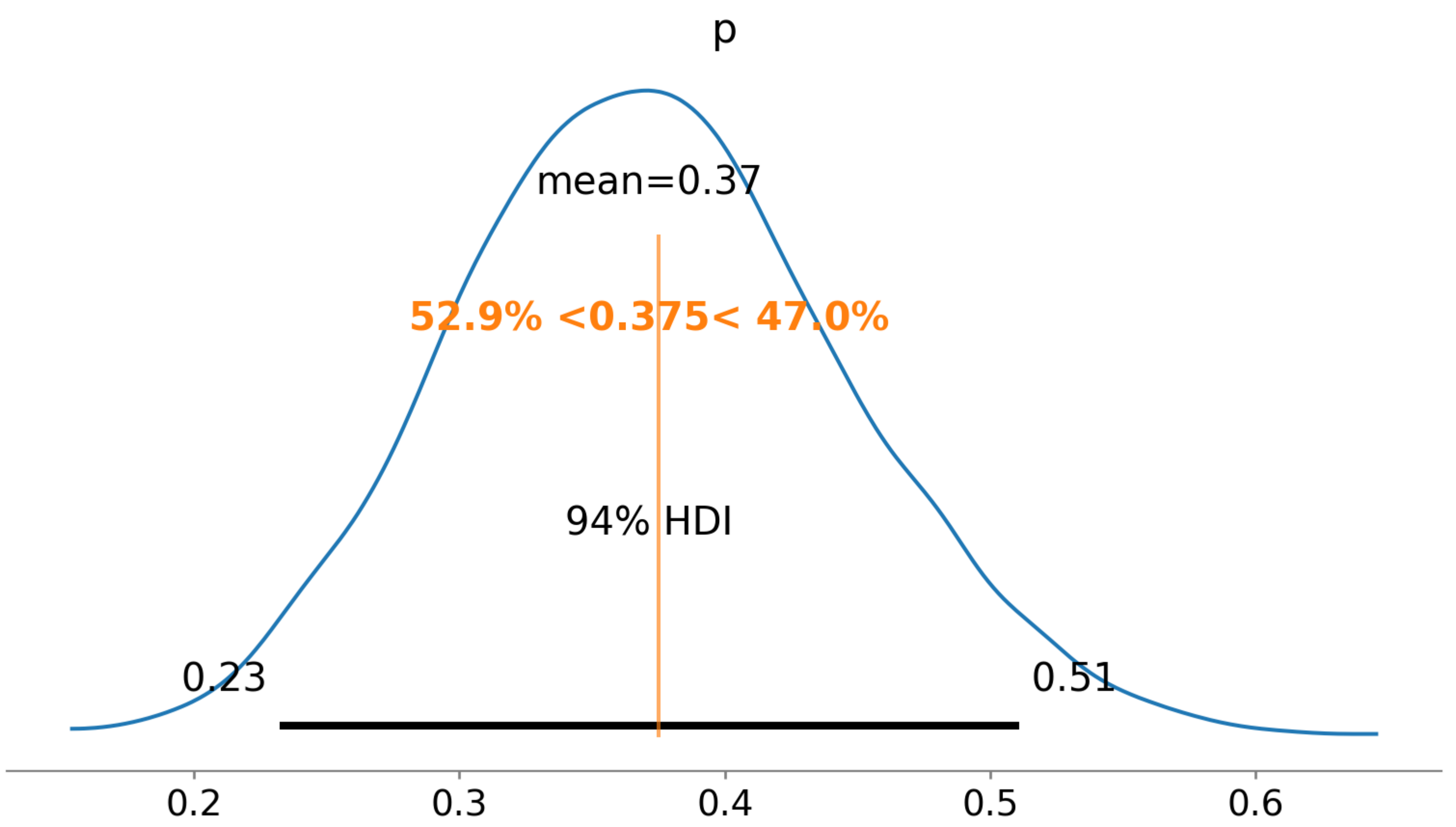
Traceplot

```
ax = az.plot_trace(trace)
plt.show()
```



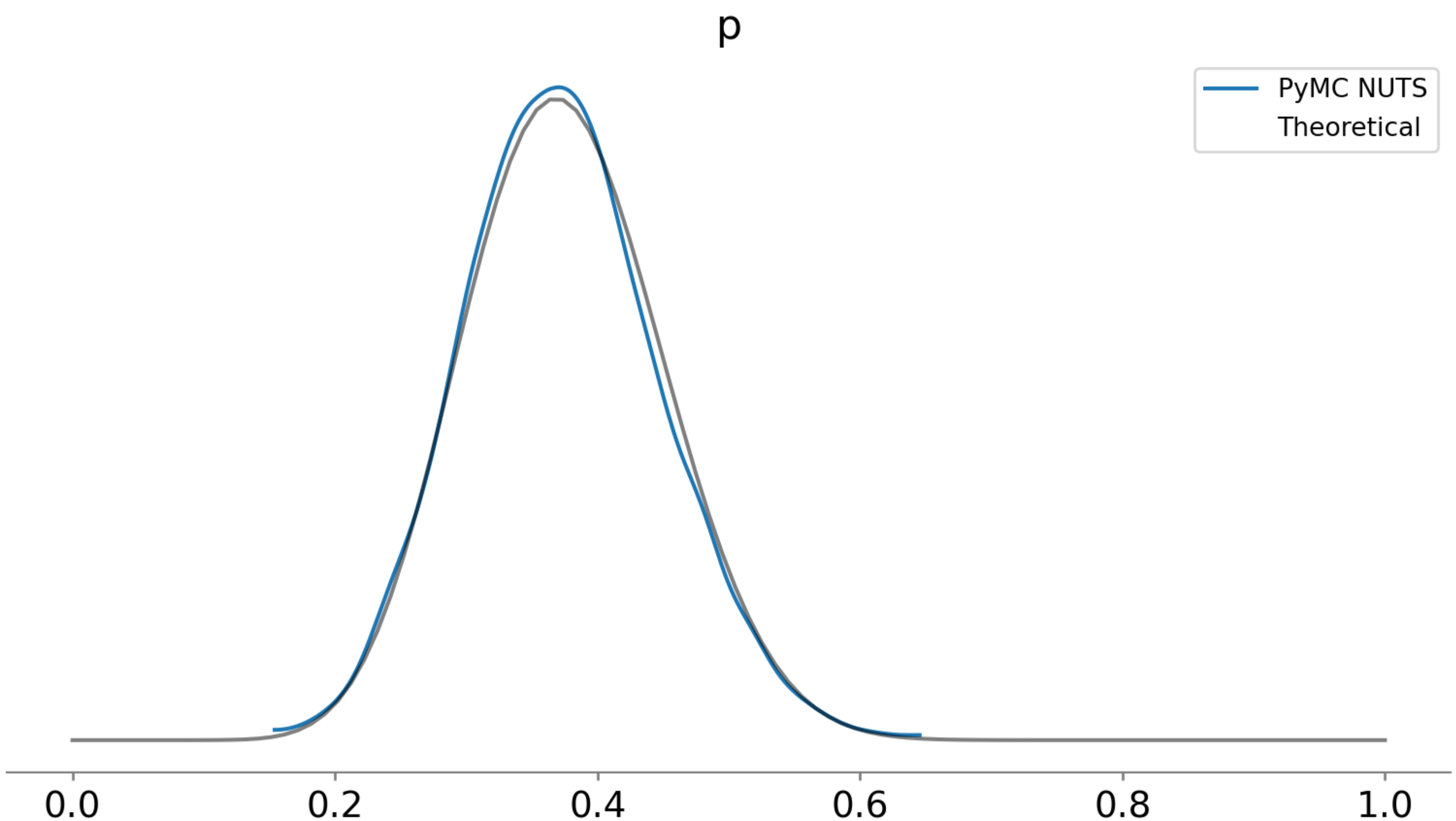
Posterior plot

```
ax = az.plot_posterior(trace, ref_val=[15/40])
plt.show()
```



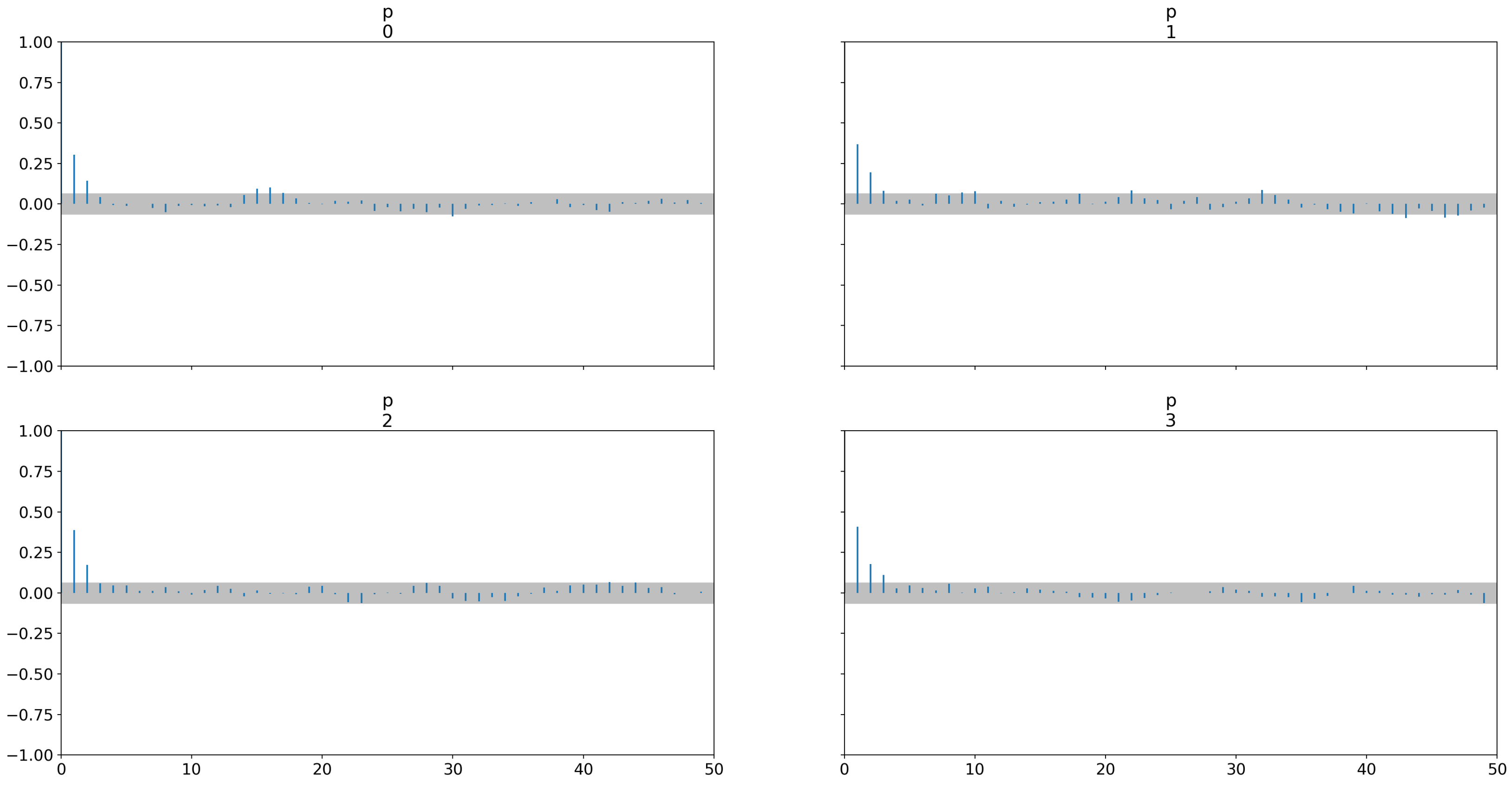
PyMC vs Theoretical

```
p = np.linspace(0, 1, 100)
post_beta = scipy.stats.beta.pdf(p,15,25)
ax = az.plot_posterior(trace, hdi_prob="hide", point_estimate=None)
plt.plot(p, post_beta, "-k", alpha=0.5, label="Theoretical")
plt.legend(['PyMC NUTS', 'Theoretical'])
plt.show()
```


p
PyMC NUTS
Theoretical
0.0
0.2
0.4
0.6
0.8
1.0

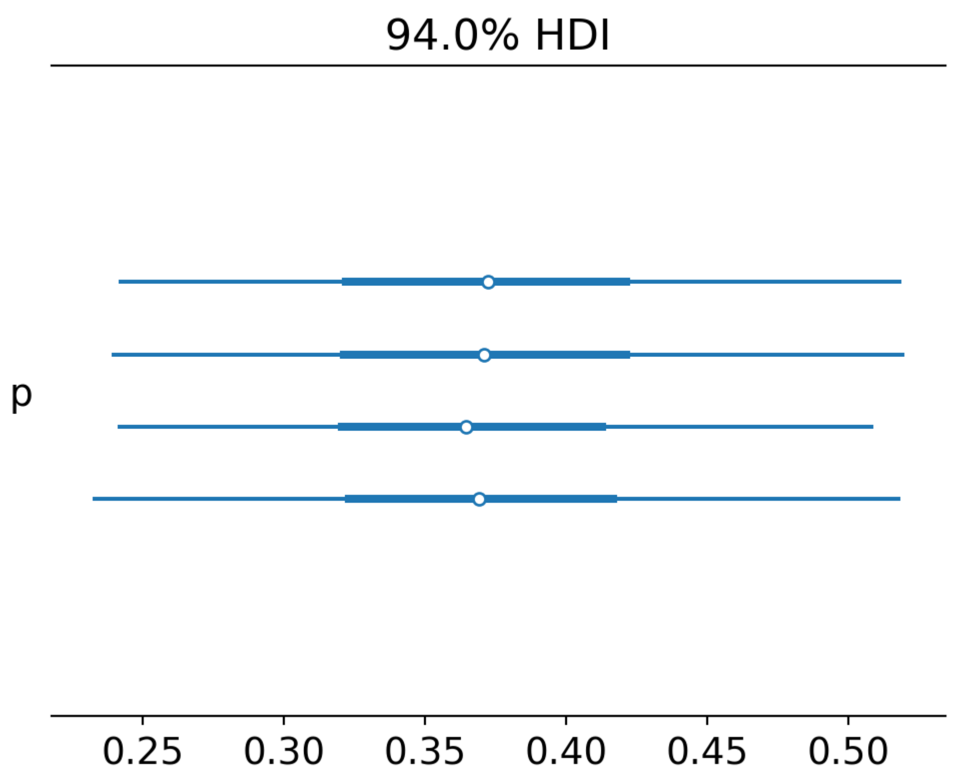
Autocorrelation plots

```
ax = az.plot_autocorr(trace, grid=(2,2), max_lag=50)
plt.show()
```



Forest plots

```
ax = az.plot_forest(trace)
plt.show()
```



Other useful diagnostics

Standard MCMC diagnostic statistics are available via `summary()` from ArviZ

```python
az.summary(trace)
```

```
    mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
p  0.372  0.074   0.232    0.511      0.002    0.001    1615.0    2222.0    1.0
```

individual methods are available for each statistics,

```python
print(az.ess(trace, method="bulk"))
```

```
<xarray.Dataset>
Dimensions:  ()
Data variables:
    p        float64 1.615e+03
```

```python
print(az.ess(trace, method="tail"))
```

```
<xarray.Dataset>
Dimensions:  ()
Data variables:
    p        float64 2.222e+03
```

```python
print(az.rhat(trace))
```

```
<xarray.Dataset>
Dimensions:  ()
Data variables:
    p        float64 1.002
```

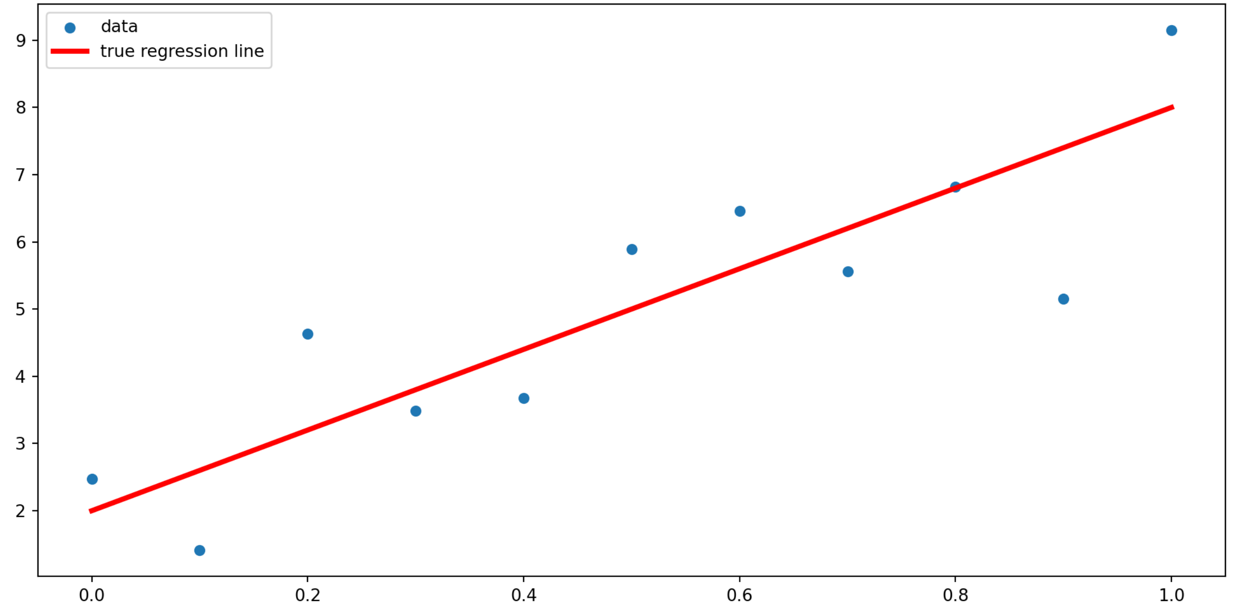
```python
print(az.mcse(trace))
```

```
<xarray.Dataset>
Dimensions:  ()
Data variables:
    p        float64 0.001844
```

Demo 1 - Linear regression

Given the below data, we want to fit a linear regression model to the following synthetic data,

```
np.random.seed(1234)
n = 11
m = 6
b = 2
x = np.linspace(0, 1, n)
y = m*x + b + np.random.randn(n)
```



Model

```
with pm.Model() as lm:
  m = pm.Normal('m', mu=0, sigma=50)
  b = pm.Normal('b', mu=0, sigma=50)
  sigma = pm.HalfNormal('sigma', sigma=5)

  pm.Normal('y', mu=m*x + b, sigma=sigma, observed=y)

  trace = pm.sample(progressbar=False, random_seed=1234)
```

```
y
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [m, b, sigma]
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws t
```

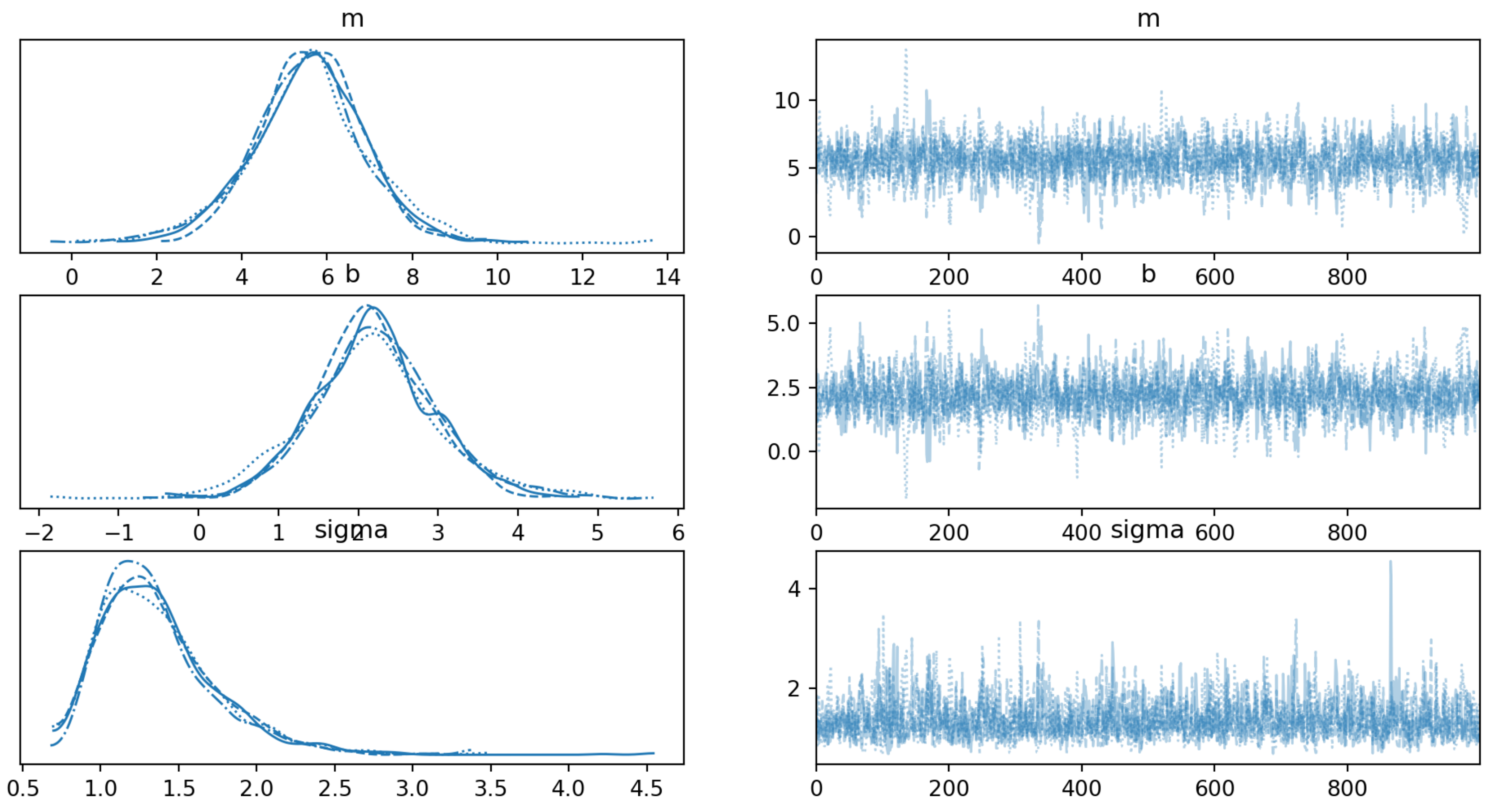
Posterior summary

```
1  az.summary(trace)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
m	5.573	1.344	2.969	8.042	0.040	0.028	1182.0	1229.0	1.0
b	2.189	0.791	0.659	3.624	0.024	0.018	1106.0	1153.0	1.0
sigma	1.366	0.378	0.788	2.075	0.010	0.007	1452.0	1722.0	1.0

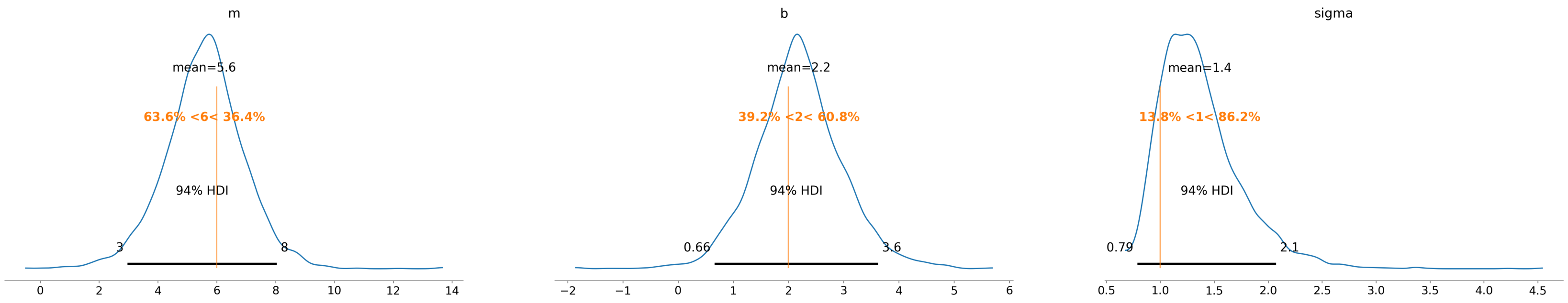
Trace plots

```
ax = az.plot_trace(trace)
plt.show()
```



Posterior plots

```
ax = az.plot_posterior(trace, ref_val=[6,2,1], grid=(1,3))
plt.show()
```

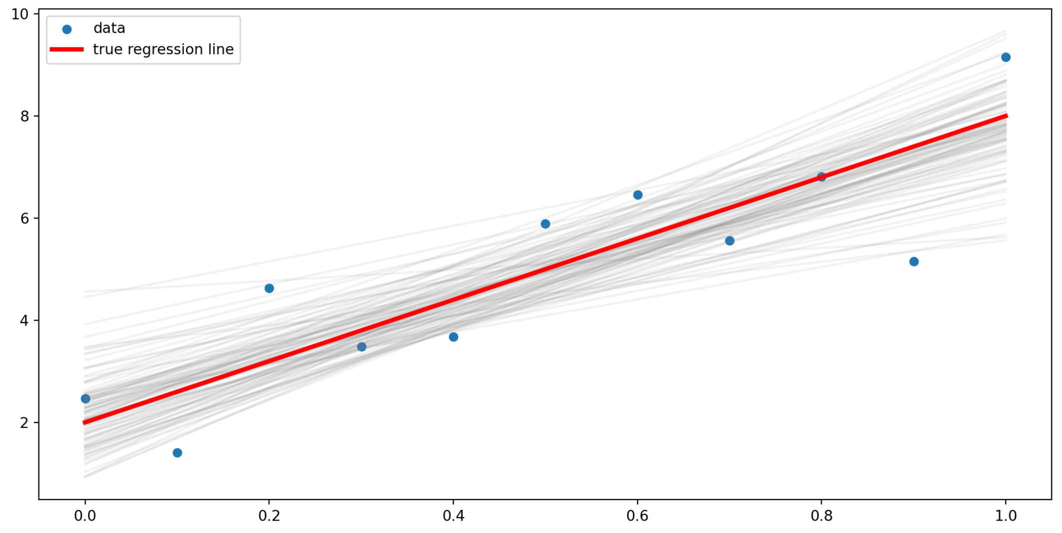


Regression line posterior draws

```
plt.scatter(x, y, s=30, label='data')

post_m = trace.posterior['m'].sel(chain=0, draw=slice(0,None,10))
post_b = trace.posterior['b'].sel(chain=0, draw=slice(0,None,10))

plt.figure(layout="constrained")
plt.scatter(x, y, s=30, label='data')
for m, b in zip(post_m.values, post_b.values):
    plt.plot(x, m*x + b, c='gray', alpha=0.1)
plt.plot(x, 6*x + 2, label='true regression line', lw=3., c='red')
plt.legend(loc='best')
plt.show()
```



Posterior predictive draws

Draws for observed variables can also be generated (posterior predictive draws) via the `sample_posterior_predictive()` method.

```
with lm:
  pp = pm.sample_posterior_predictive(trace, progressbar=False)
```

```
Sampling: [y]
```

```
pp
```

arviz.InferenceData

▸ posterior_predictive

▸ observed_data

```
pp.posterior_predictive
```

```
<xarray.Dataset>
Dimensions:  (chain: 4, draw: 1000, y_dim_2: 11)
Coordinates:
  * chain    (chain) int64 0 1 2 3
  * draw     (draw) int64 0 1 2 3 4 5 6 7 8 ... 992 993 994 995 996 997 998 999
  * y_dim_2  (y_dim_2) int64 0 1 2 3 4 5 6 7 8 9 10
Data variables:
    y        (chain, draw, y_dim_2) float64 2.857 2.19 4.669 ... 7.195 5.406
Attributes:
    created_at:                 2023-03-24T16:28:16.241911
```

```
    arviz_version:              0.15.1
    inference_library:          pymc
    inference_library_version:  5.1.2
```

xarray.Dataset

▸ Dimensions:

(**chain**: 4, **draw**: 1000, **y_dim_2**: 11)

▾ Coordinates:

chain
(chain)
int64
0 1 2 3

draw
(draw)
int64
0 1 2 3 4 5 ... 995 996 997 998 999

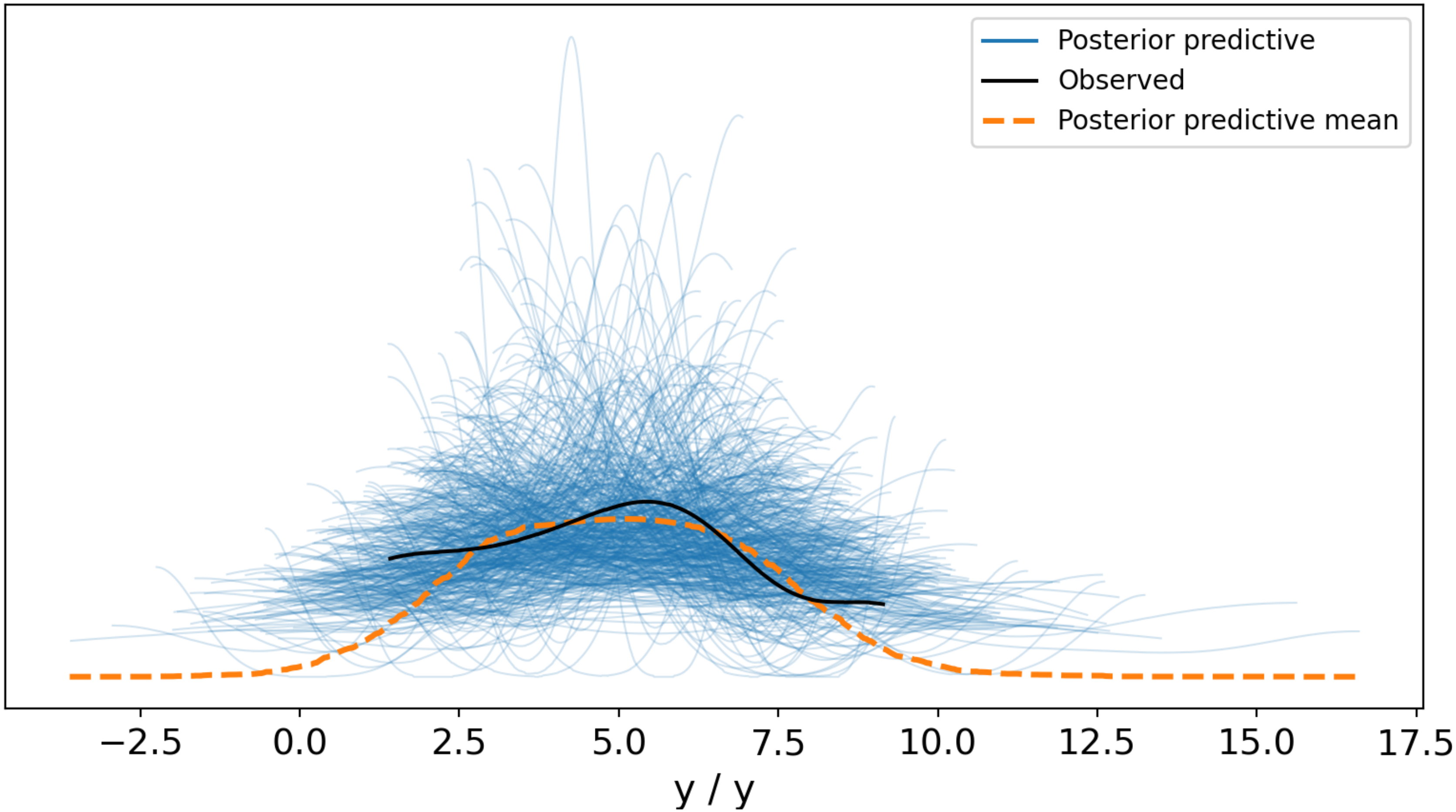
y_dim_2
(y_dim_2)
int64
0 1 2 3 4 5 6 7 8 9 10

▾ Data variables:

y
(chain, draw, y_dim_2)
float64
2.857 2.19 4.669 ... 7.195 5.406

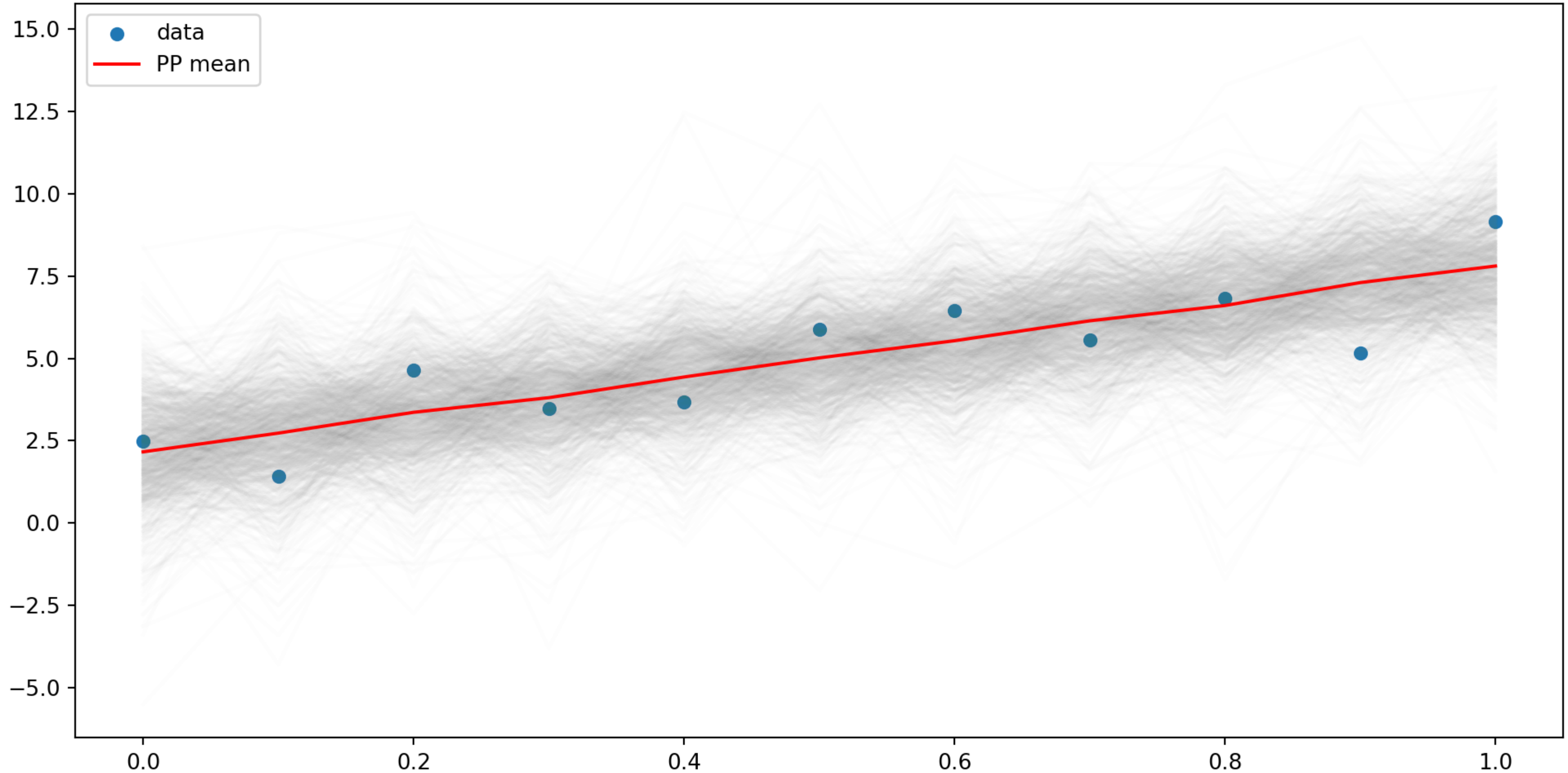
Plotting the posterior predictive distribution

```
az.plot_ppc(pp, num_pp_samples=500)
plt.show()
```



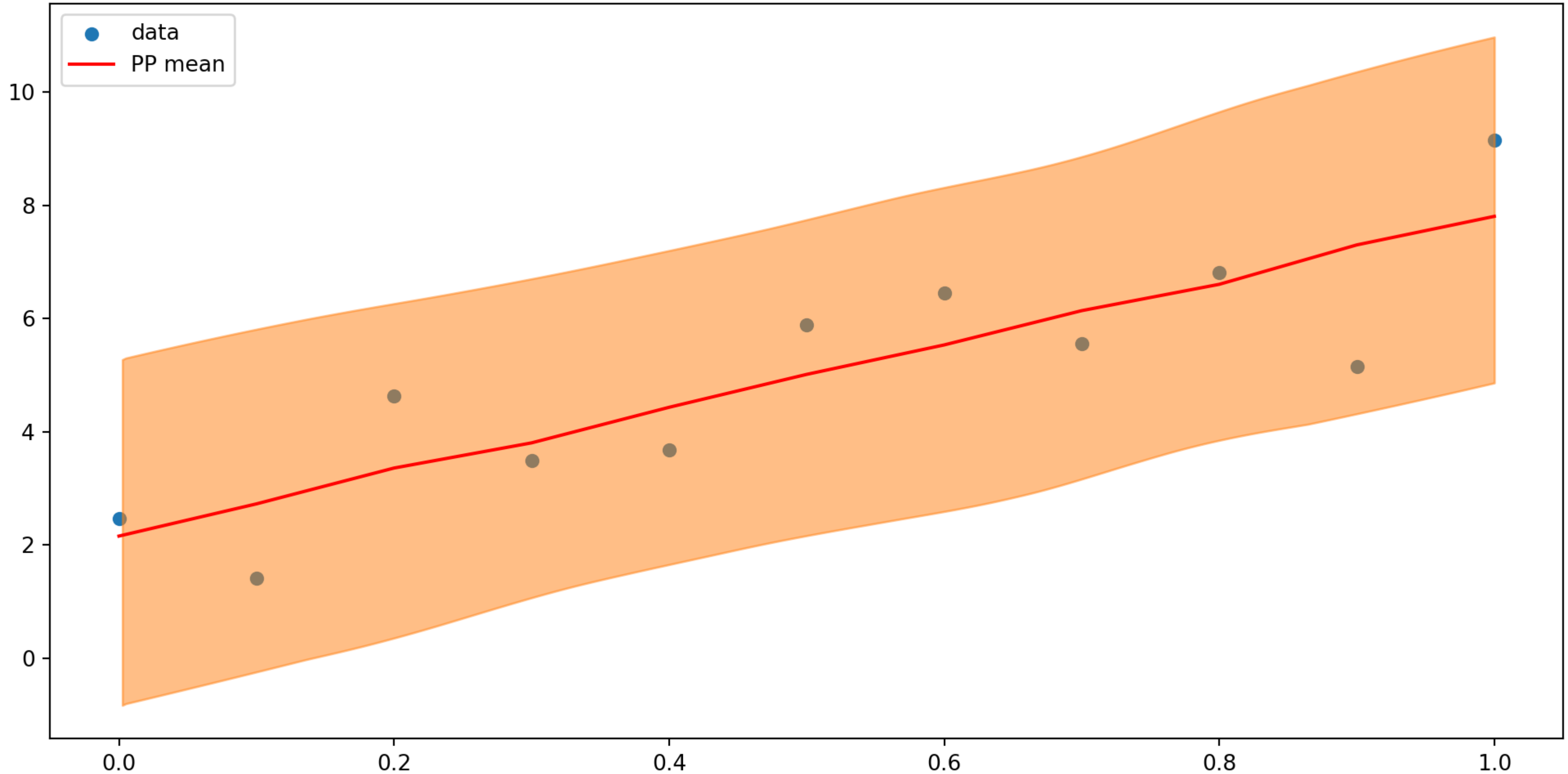
PP draws

```
plt.figure(layout="constrained")
plt.scatter(x, y, s=30, label='data')
plt.plot(x, pp.posterior_predictive['y'].sel(chain=0).T, c="grey", alpha=0.01)
plt.plot(x, np.mean(pp.posterior_predictive['y'].sel(chain=0).T, axis=1), c='red', label="PP mean")
plt.legend()
plt.show()
```


data
PP mean
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
0.0
0.2
0.4
0.6
0.8
1.0

PP HDI

```
plt.figure(layout="constrained")
plt.scatter(x, y, s=30, label='data')
plt.plot(x, np.mean(pp.posterior_predictive['y'].sel(chain=0).T, axis=1), c='red', label="PP mean")
az.plot_hdi(x, pp.posterior_predictive['y'])
plt.legend()
plt.show()
```



data
PP mean
10
8
6
4
2
0
0.0
0.2
0.4
0.6
0.8
1.0

Model revision

```
with pm.Model() as lm2:
  m = pm.Normal('m', mu=0, sigma=50)
  b = pm.Normal('b', mu=0, sigma=50)
  sigma = pm.HalfNormal('sigma', sigma=5)

  y_hat = pm.Deterministic("y_hat", m*x + b)

  pm.Normal('y', mu=y_hat, sigma=sigma, observed=y)

  trace = pm.sample(random_seed=1234, progressbar=False)
  pp = pm.sample_posterior_predictive(trace, var_names=["y_hat"], progressbar=False)
```

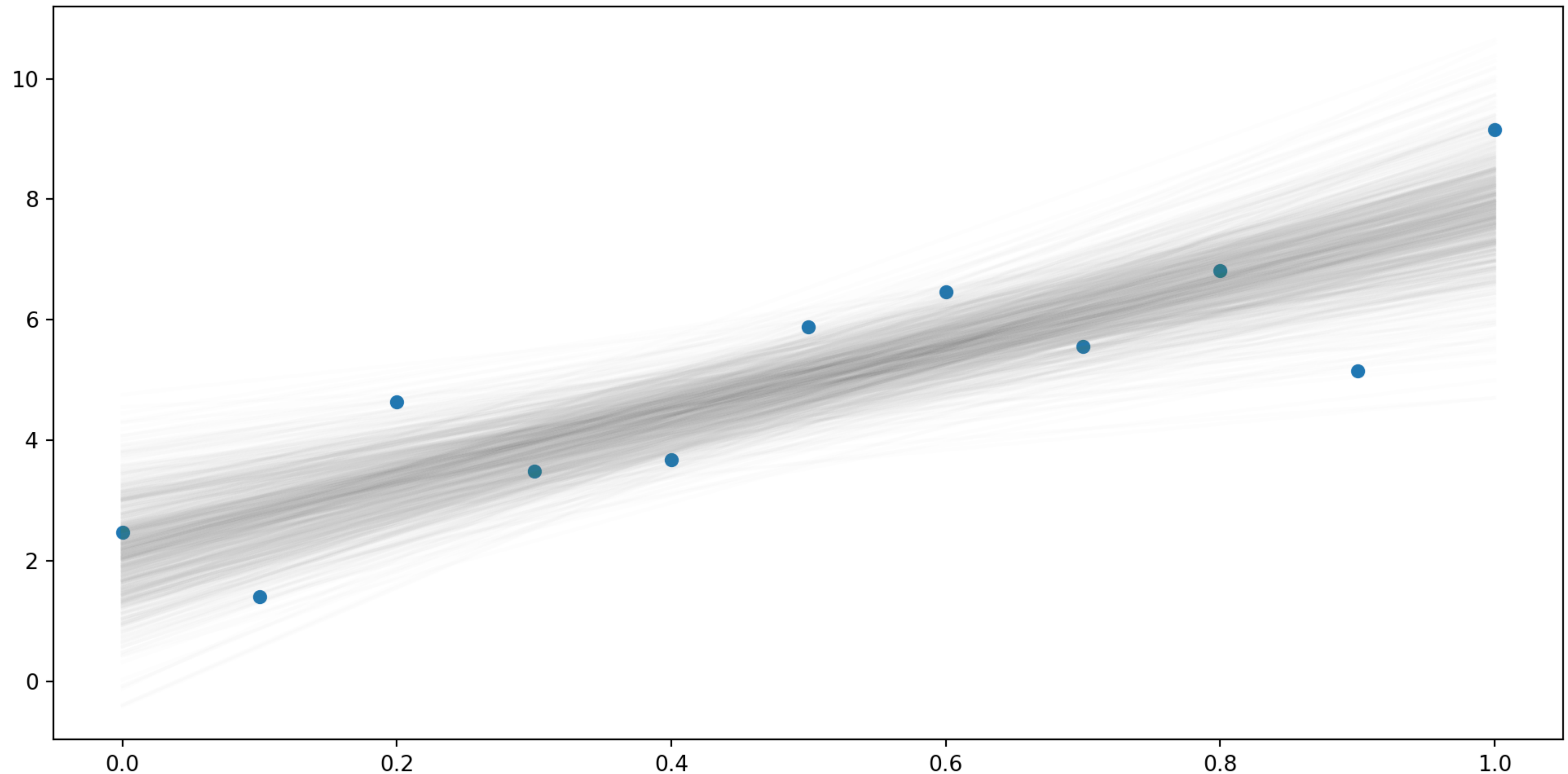
```
y

Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [m, b, sigma]
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 1 seconds.
Sampling: []
```

```
pm.summary(trace)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
m	5.573	1.344	2.969	8.042	0.040	0.028	1182.0	1229.0	1.0
b	2.189	0.791	0.659	3.624	0.024	0.018	1106.0	1153.0	1.0
sigma	1.366	0.378	0.788	2.075	0.010	0.007	1452.0	1722.0	1.0
y_hat[0]	2.189	0.791	0.659	3.624	0.024	0.018	1106.0	1153.0	1.0
y_hat[1]	2.746	0.681	1.427	4.009	0.021	0.015	1155.0	1245.0	1.0
y_hat[2]	3.303	0.583	2.228	4.424	0.017	0.012	1254.0	1319.0	1.0
y_hat[3]	3.861	0.501	2.933	4.821	0.013	0.010	1481.0	1786.0	1.0
y_hat[4]	4.418	0.446	3.543	5.218	0.010	0.007	2031.0	2364.0	1.0
y_hat[5]	4.975	0.427	4.187	5.791	0.007	0.005	3266.0	2804.0	1.0
y_hat[6]	5.532	0.449	4.671	6.389	0.007	0.005	4478.0	2810.0	1.0
y_hat[7]	6.090	0.508	5.090	7.017	0.008	0.006	3929.0	2889.0	1.0
y_hat[8]	6.647	0.591	5.568	7.810	0.011	0.008	2963.0	2707.0	1.0
y_hat[9]	7.204	0.691	5.924	8.542	0.014	0.010	2389.0	2337.0	1.0
y_hat[10]	7.761	0.801	6.358	9.405	0.018	0.013	2087.0	2291.0	1.0

```
plt.figure(layout="constrained")
plt.plot(x, pp.posterior_predictive['y_hat'].sel(chain=0).T, c="grey", alpha=0.01)
plt.scatter(x, y, s=30, label='data')
plt.show()
```



Demo 2 - Bayesian Lasso

```
n = 50
k = 100

np.random.seed(1234)
X = np.random.normal(size=(n, k))

beta = np.zeros(shape=k)
beta[[10,30,50,70]] =  10
beta[[20,40,60,80]] = -10

y = X @ beta + np.random.normal(size=n)
```


Based on Bayesian Sparse Regression and Lasso regression with block updating

Naive model

```
with pm.Model() as bayes_naive:
  b = pm.Flat("beta", shape=k)
  s = pm.HalfNormal('sigma', sigma=2)

  pm.Normal("y", mu=X @ b, sigma=s, observed=y)

  trace = pm.sample(progressbar=False, random_seed=12345)
```

```
y
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [beta, sigma]
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 70 seconds.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during sampling. See htt
   arxiv.org/abs/1903.08008 for details
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is needed for
   rhat and ess computation. See https://arxiv.org/abs/1903.08008 for details
There were 113 divergences after tuning. Increase `target_accept` or reparameterize.
Chain 0 reached the maximum tree depth. Increase `max_treedepth`, increase `target_accept` or reparameterize
Chain 1 reached the maximum tree depth. Increase `max_treedepth`, increase `target_accept` or reparameterize
Chain 2 reached the maximum tree depth. Increase `max_treedepth`, increase `target_accept` or reparameterize
Chain 3 reached the maximum tree depth. Increase `max_treedepth`, increase `target_accept` or reparameterize
```

```
az.summary(trace)
```

```
           mean       sd    hdi_3%    hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail
beta[0]   643.620   606.139  -622.272  1402.288    261.586  196.065       6.0      14.0
beta[1]  -410.200   341.106 -1078.665   124.358    101.833   76.145      12.0      20.0
beta[2]   717.738  1008.215  -680.198  2520.985    482.446  376.947       5.0      12.0
beta[3]  -519.149  1119.439 -2671.450  1043.628    549.247  419.469       5.0      12.0
beta[4]  1471.100   990.544  -179.163  2808.723    471.027  364.160       5.0      21.0
...           ...       ...       ...       ...        ...      ...       ...       ...
beta[96]  420.720  1056.327 -1260.449  2382.936    487.813  369.004       5.0      12.0
beta[97] -143.535  1203.422 -2170.522  1927.823    574.093  436.428       5.0      14.0
beta[98] -818.080   685.229 -2423.653   190.270    313.078  245.187       6.0      12.0
beta[99]  261.936   912.018 -1070.744  1629.130    440.188  335.261       5.0      15.0
sigma       2.432     1.316     0.461     4.800      0.467    0.343       7.0      27.0

[101 rows x 9 columns]
```

Weakly informative model

```
with pm.Model() as bayes_weak:
  b = pm.Normal("beta", mu=0, sigma=10, shape=k)
  s = pm.HalfNormal('sigma', sigma=2)

  pm.Normal("y", mu=X @ b, sigma=s, observed=y)

  trace = pm.sample(progressbar=False, random_seed=12345)
```

```
y
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [beta, sigma]
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 71 seconds.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during sampling. See htt
   arxiv.org/abs/1903.08008 for details
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is needed for
   rhat and ess computation. See https://arxiv.org/abs/1903.08008 for details
There were 40 divergences after tuning. Increase `target_accept` or reparameterize.
Chain 2 reached the maximum tree depth. Increase `max_treedepth`, increase `target_accept` or reparameterize
```

```python
az.summary(trace)
```

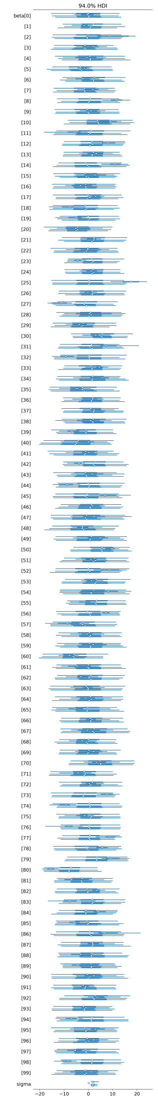
```
           mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
beta[0]  -0.077  6.620 -12.749   12.799      0.200    0.548     830.0    1795.0   1.07
beta[1]   0.527  5.875 -10.243   12.733      0.693    0.525      63.0    2169.0   1.13
beta[2]   2.191  8.614 -11.691   19.642      2.457    1.780      13.0      21.0   1.23
beta[3]  -0.756  6.533 -14.000   10.236      0.958    0.681      43.0    1570.0   1.06
beta[4]   0.399  7.131 -12.064   14.597      1.102    0.785      40.0    1274.0   1.06
...         ...    ...     ...      ...        ...      ...       ...       ...    ...
beta[96]  0.085  6.255 -10.955   12.770      0.510    0.462     162.0    1586.0   1.03
beta[97] -2.220  6.716 -14.092   11.428      1.059    0.754      40.0     234.0   1.07
beta[98] -1.101  7.490 -14.253   11.607      2.305    1.677      10.0      31.0   1.29
beta[99] -1.003  6.763 -14.138   10.319      0.633    0.449     125.0    1170.0   1.02
sigma     1.606  1.172   0.013    3.646      0.393    0.288       7.0      11.0   1.57

[101 rows x 9 columns]
```

```python
1 az.summary(trace).iloc[[10,20,30,40,50,60,70,80]]
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
beta[10]	4.967	6.752	-9.493	15.785	1.221	0.872	31.0	2230.0	1.09
beta[20]	-4.413	6.705	-15.987	8.051	1.157	0.825	34.0	1600.0	1.08
beta[30]	4.986	5.798	-6.198	15.950	0.606	0.560	95.0	1727.0	1.06
beta[40]	-3.742	7.697	-19.201	9.502	1.128	0.802	48.0	263.0	1.05
beta[50]	6.186	6.122	-6.222	16.288	0.635	0.451	122.0	1311.0	1.04
beta[60]	-6.317	6.212	-17.585	6.570	0.327	0.231	364.0	1647.0	1.10
beta[70]	4.856	6.688	-6.852	18.421	0.461	0.423	217.0	1495.0	1.04
beta[80]	-9.648	6.434	-19.370	2.766	1.712	1.256	15.0	217.0	1.18

```python
ax = az.plot_forest(trace)
plt.tight_layout()
plt.show()
```



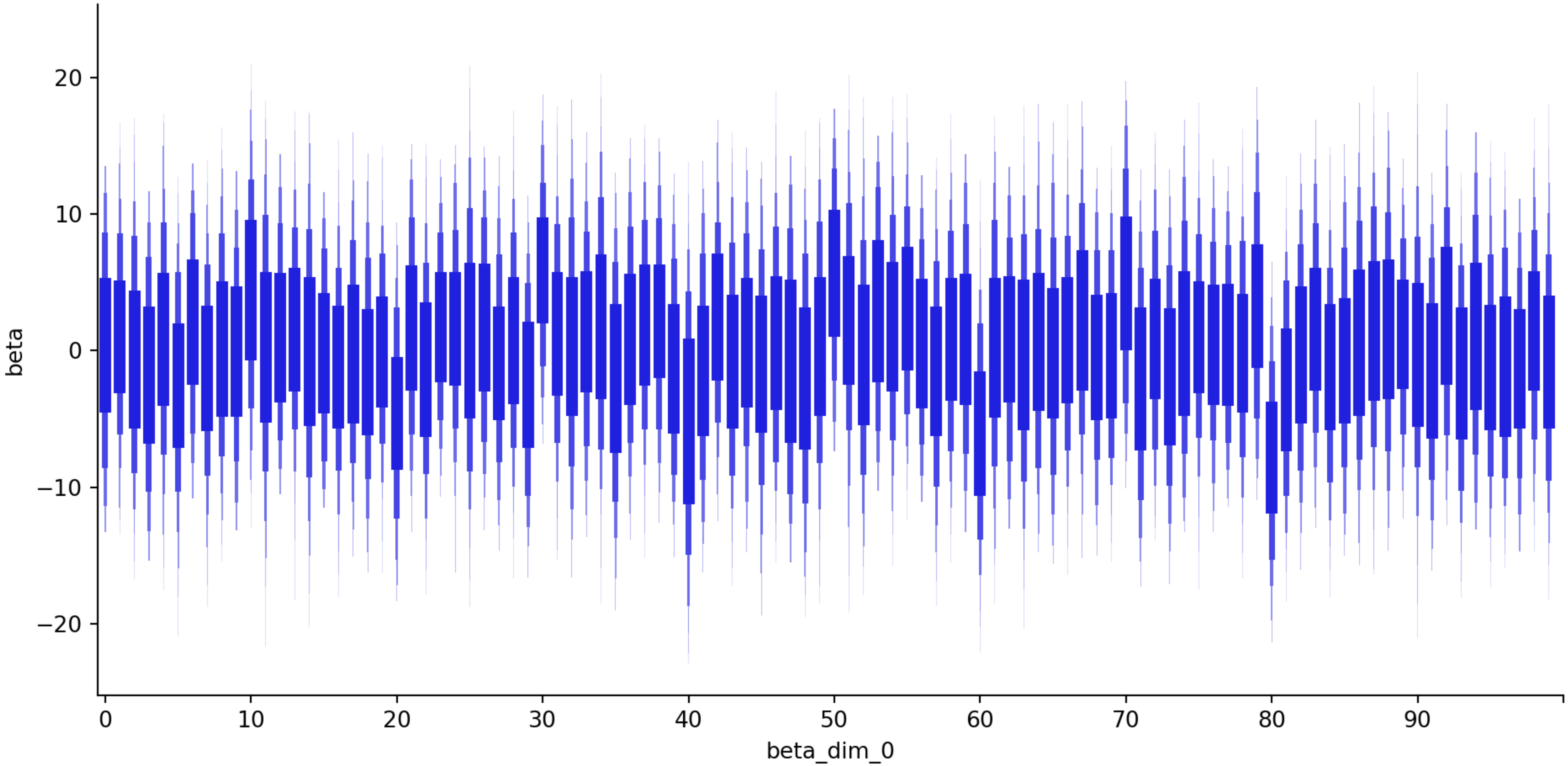
Plot helper

```
def plot_slope(trace, prior="beta", chain=0):
  post = (trace.posterior[prior]
          .to_dataframe()
          .reset_index()
          .query(f"chain == {chain}")
         )

  sns.catplot(x="beta_dim_0", y="beta", data=post, kind="boxen", linewidth=0, color='blue', aspect=2, s
  plt.tight_layout()
  plt.xticks(range(0,110,10))
  plt.show()

```

```
1 plot_slope(trace)
```



Laplace Prior

```
with pm.Model() as bayes_lasso:
  b = pm.Laplace("beta", 0, 1, shape=k)
  s = pm.HalfNormal('sigma', sigma=1)

  pm.Normal("y", mu=X @ b, sigma=s, observed=y)

  trace = pm.sample(progressbar=False, random_seed=1234)
```

```
y
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [beta, sigma]
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 23 seconds.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during sampling. See htt
   arxiv.org/abs/1903.08008 for details
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is needed for
   rhat and ess computation. See https://arxiv.org/abs/1903.08008 for details
There were 239 divergences after tuning. Increase `target_accept` or reparameterize.
```

```python
az.summary(trace)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
beta[0]	0.067	0.785	-1.402	1.716	0.011	0.018	4345.0	1972.0	1.02
beta[1]	0.308	0.829	-1.179	1.710	0.152	0.119	38.0	533.0	1.07
beta[2]	-0.045	0.783	-1.634	1.478	0.012	0.015	4101.0	2327.0	1.02
beta[3]	-0.215	0.804	-1.865	1.157	0.075	0.053	95.0	2069.0	1.03
beta[4]	0.063	0.775	-1.508	1.570	0.012	0.013	4381.0	2580.0	1.11
...	...	...	...	...	...	...	...	...	...
beta[96]	0.173	0.774	-1.322	1.427	0.136	0.097	38.0	2541.0	1.07
beta[97]	-0.139	0.721	-1.532	1.351	0.011	0.014	3961.0	2368.0	1.11
beta[98]	0.244	0.708	-1.038	1.689	0.028	0.020	570.0	2040.0	1.01
beta[99]	-0.277	0.750	-1.850	1.061	0.020	0.016	1298.0	1953.0	1.01
sigma	0.825	0.522	0.171	1.790	0.129	0.093	10.0	8.0	1.30

[101 rows x 9 columns]

```python
az.summary(trace).iloc[[10,20,30,40,50,60,70,80]]
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
beta[10]	8.306	1.184	6.066	10.577	0.025	0.018	2301.0	2176.0	1.04
beta[20]	-8.298	1.253	-10.672	-5.908	0.029	0.021	1860.0	2316.0	1.01
beta[30]	8.673	0.979	6.683	10.340	0.050	0.035	551.0	2361.0	1.01
beta[40]	-8.781	1.517	-11.710	-5.931	0.031	0.022	2408.0	2496.0	1.12
beta[50]	9.076	0.981	7.097	10.773	0.041	0.029	1055.0	2812.0	1.01
beta[60]	-9.313	1.133	-11.377	-7.166	0.133	0.098	69.0	2392.0	1.04
beta[70]	8.636	1.228	6.214	10.546	0.167	0.125	53.0	2107.0	1.05
beta[80]	-9.984	0.888	-11.761	-8.366	0.017	0.012	2629.0	2425.0	1.05

```
1 plot_slope(trace)
```