PyMC + ArviZ

Lecture 23

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 PyTensor 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.

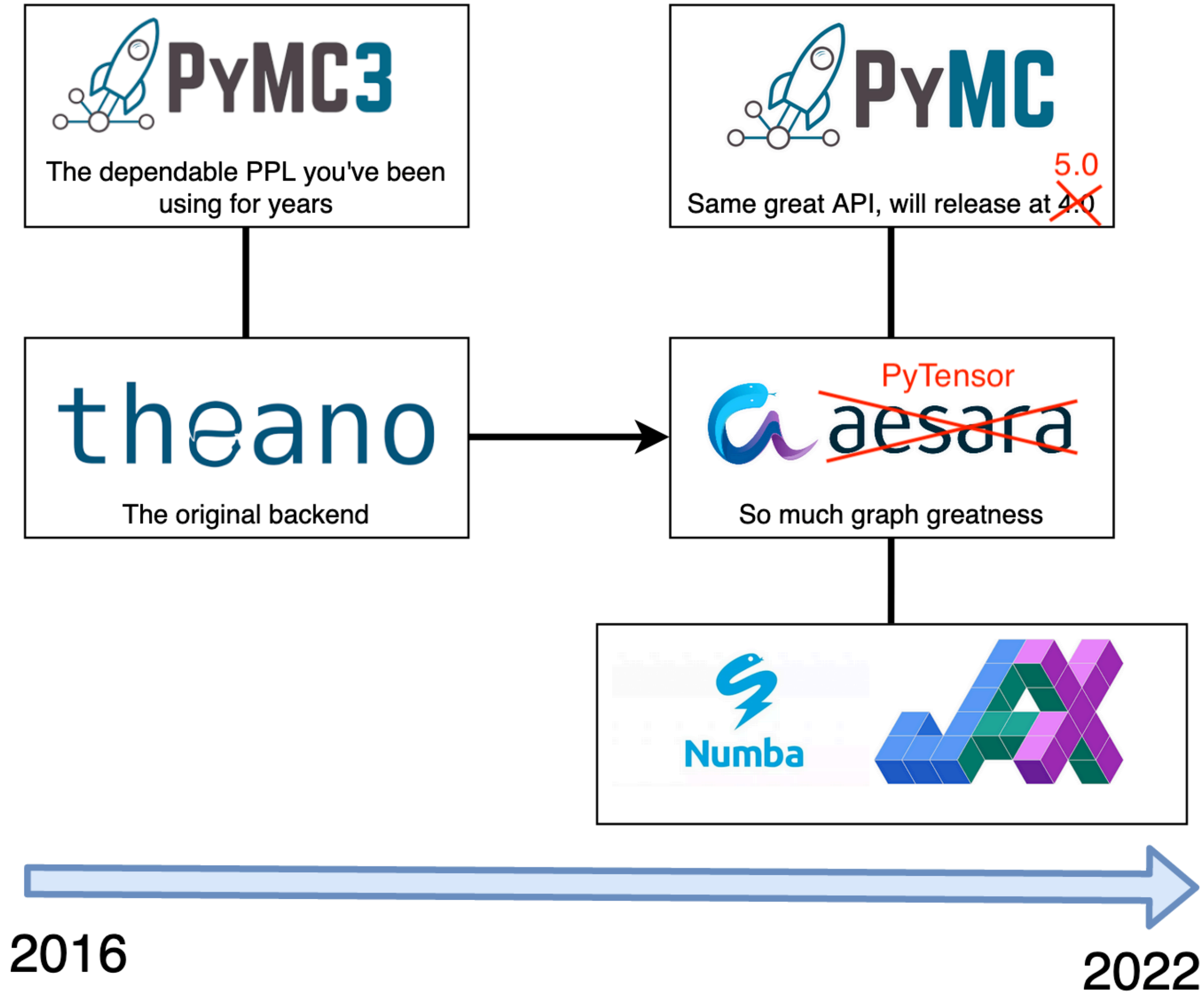
```python
import pymc as pm
```

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



Model basics

All models are derived from pymc's `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)
```

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
```

$$x \sim \text{Normal}(0,\ 1)$$

```
type(x)
```

```
pytensor.tensor.variable.TensorVariable
```

Using a component without a context

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File ~/.pyenv/versions/3.12.3/lib/python3.12/site-packages/pymc/distributions/distribution.py:481,
    479     from pymc.model import Model
--> 481     model = Model.get_context()
    482 except TypeError:

File ~/.pyenv/versions/3.12.3/lib/python3.12/site-packages/pymc/model/core.py:514, in Model.get_co
    513 if model is None and error_if_none:
--> 514     raise TypeError("No model on context stack")
    515 return model

TypeError: No model on context stack

During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 x = pm.Normal(    , mu=0, sigma=1)

File ~/.pyenv/versions/3.12.3/lib/python3.12/site-packages/pymc/distributions/distribution.py:483,
    481     model = Model.get_context()
    482 except TypeError:
```

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 they do have some useful methods and attributes:

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

```
pytensor.tensor.variable.TensorVariable
```

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

```
NormalRV(name=normal,signature=(),()->(),dtype=float64,inplace=False)
```

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

```
array(0.19112)
```

Standalone RVs

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

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

```
z
```

```
normal_rv{"(),()->()"}.out
```

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

```
array([[ 2.36473,  2.77069, -3.95415],
       [-0.54568,  0.05946, -0.46267]])
```

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 ~ Normal(0, 1), y ~ Normal(x, 1)]
```

Variable heirarchy

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

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

```
array([ 0.93819, -1.54829, -2.31653])
```

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

```
array([[-0.24739,  1.96912,  0.92891],
       [-0.94138, -1.1763 ,  1.52653],
       [-0.60988, -0.95977, -1.36152],
       ...,
       [-0.57515,  1.24518,  0.95842],
       [ 3.64309,  2.79414,  4.08703],
       [ 0.93813,  1.46746,  1.99498]], sh
```

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

```
np.float64(0.08511372203814124)
```

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

```
np.float64(2.013552990011018)
```

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

```
np.float64(1.4189971775874333)
```

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)
```

```
beta_binom.basic_RVs
```

```
[p ~ Beta(10, 10), x ~ Binomial(20, p)]
```

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)
```

`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.

Digging into `trace`

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

```
<xarray.Dataset> Size: 40kB
Dimensions:  (chain: 4, draw: 1000)
Coordinates:
  * chain    (chain) int64 32B 0 1 2 3
  * draw     (draw) int64 8kB 0 1 2 3 4 5 6 7 ... 993 994 995 996 997 998 999
Data variables:
    p        (chain, draw) float64 32kB 0.3347 0.3435 0.2629 ... 0.3276 0.3486
Attributes:
    created_at:                 2025-04-11T13:49:19.633849+00:00
    arviz_version:              0.21.0
    inference_library:          pymc
    inference_library_version:  5.22.0
    sampling_time:              0.2469632625579834
    tuning_steps:               1000
```

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

```
(4, 1000)
```

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

```
(1000,)
```

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

```
(4, 50)
```

As a DataFrame

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

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

		p
chain	**draw**	
0	0	0.334673
	1	0.343498
	2	0.262910
	3	0.346714
	4	0.288526
...	...	...
3	995	0.421379
	996	0.432828
	997	0.327570
	998	0.327570
	999	0.348614

4000 rows × 1 columns

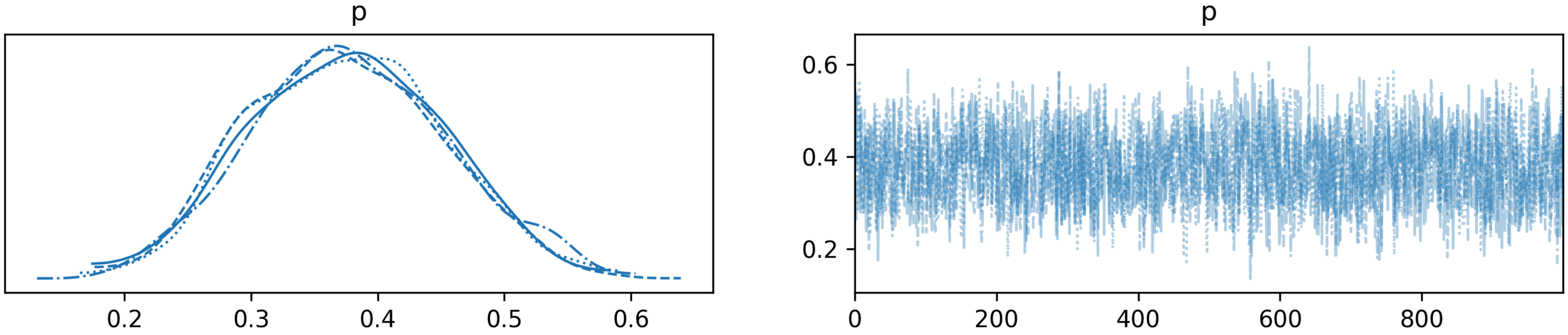
```python
trace.posterior["p"][0,:].to_dataframe()
```

	chain	p
draw		
0	0	0.334673
1	0	0.343498
2	0	0.262910
3	0	0.346714
4	0	0.288526
...	...	...
995	0	0.226934
996	0	0.483832
997	0	0.251825
998	0	0.486013
999	0	0.282455

1000 rows × 2 columns

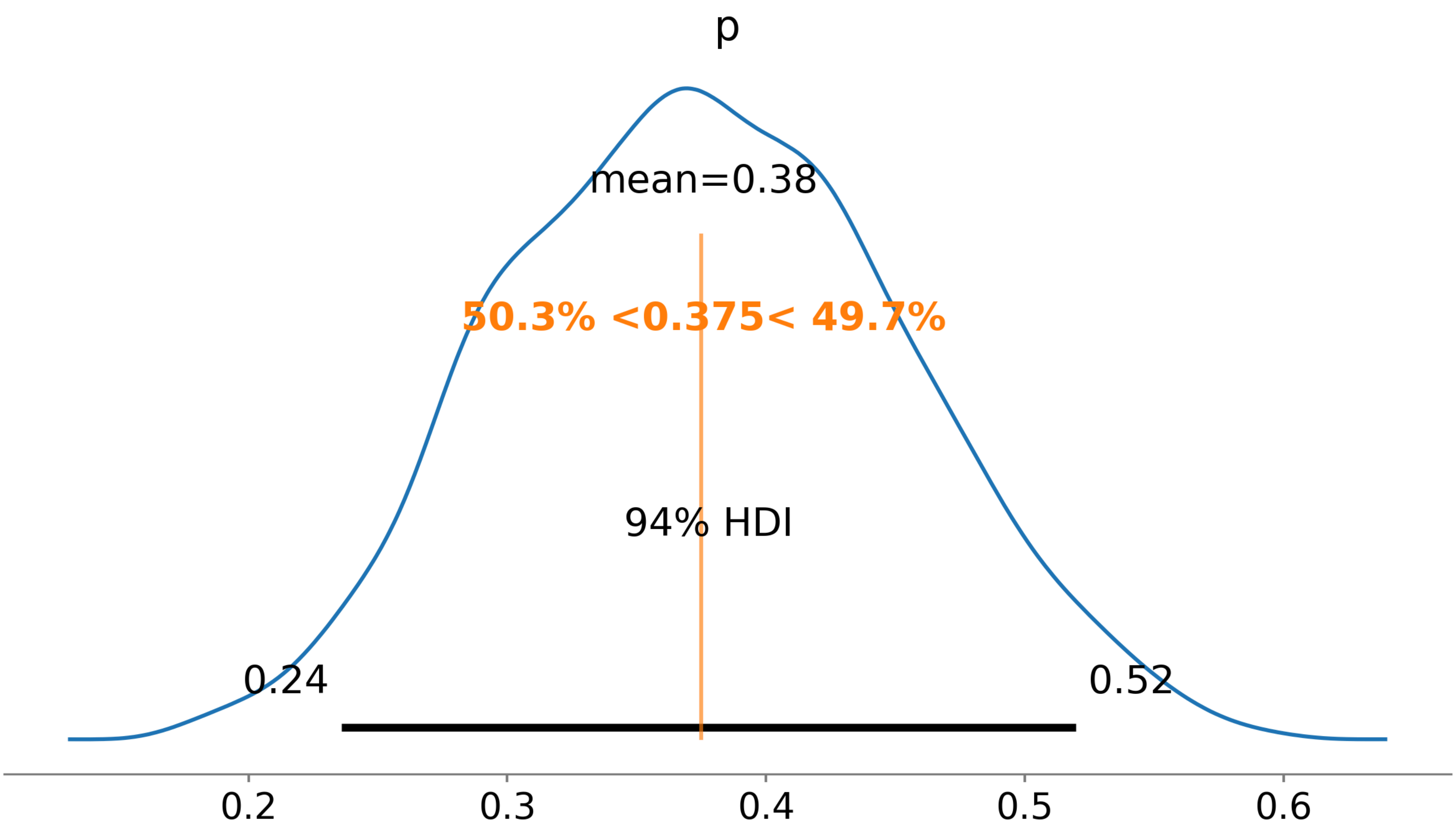
Traceplots with ArviZ

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

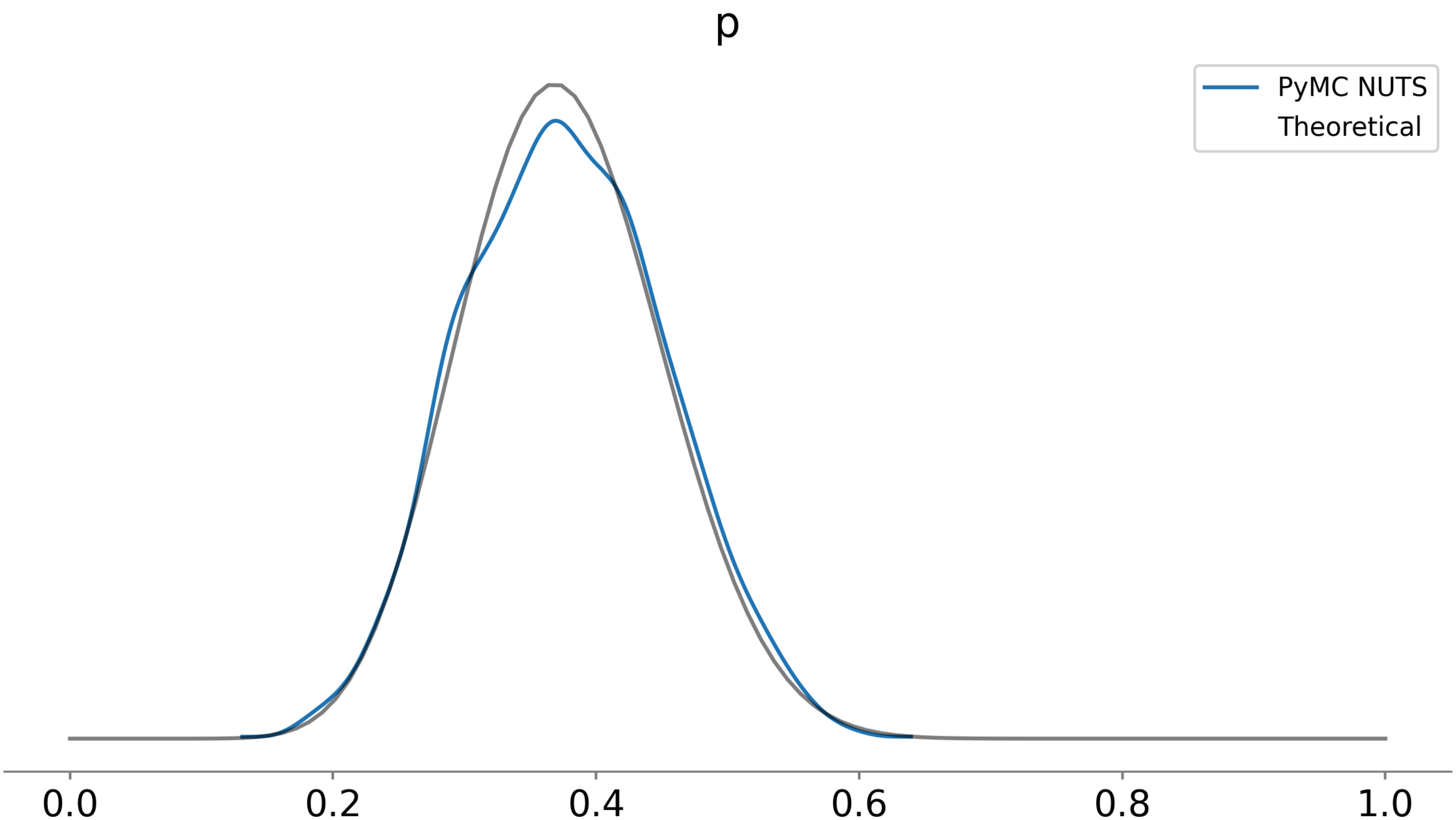


Posterior plot with ArviZ

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

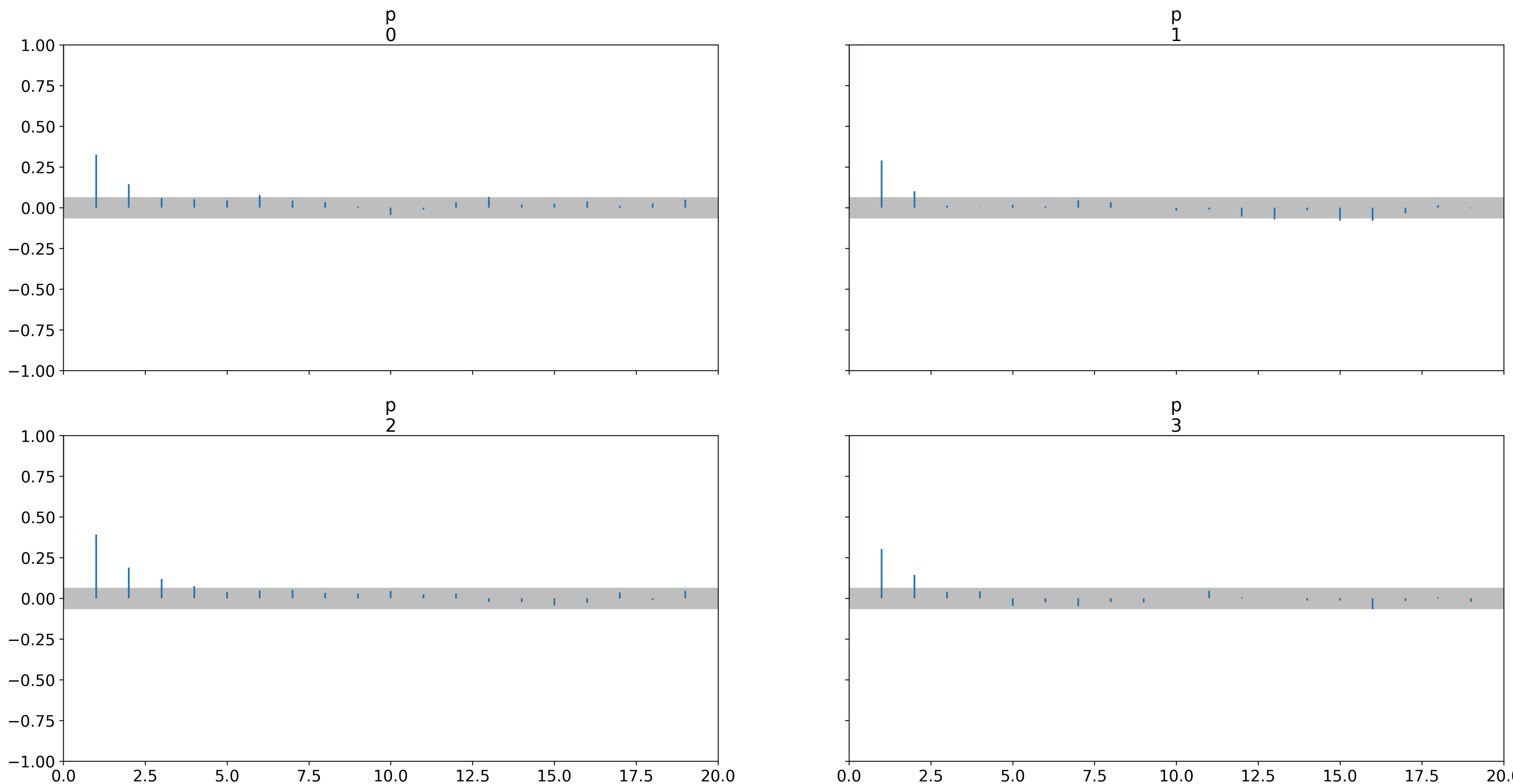

p
mean=0.38
50.3% <0.375< 49.7%
94% HDI
0.24
0.52
0.2
0.3
0.4
0.5
0.6

PyMC vs Theoretical



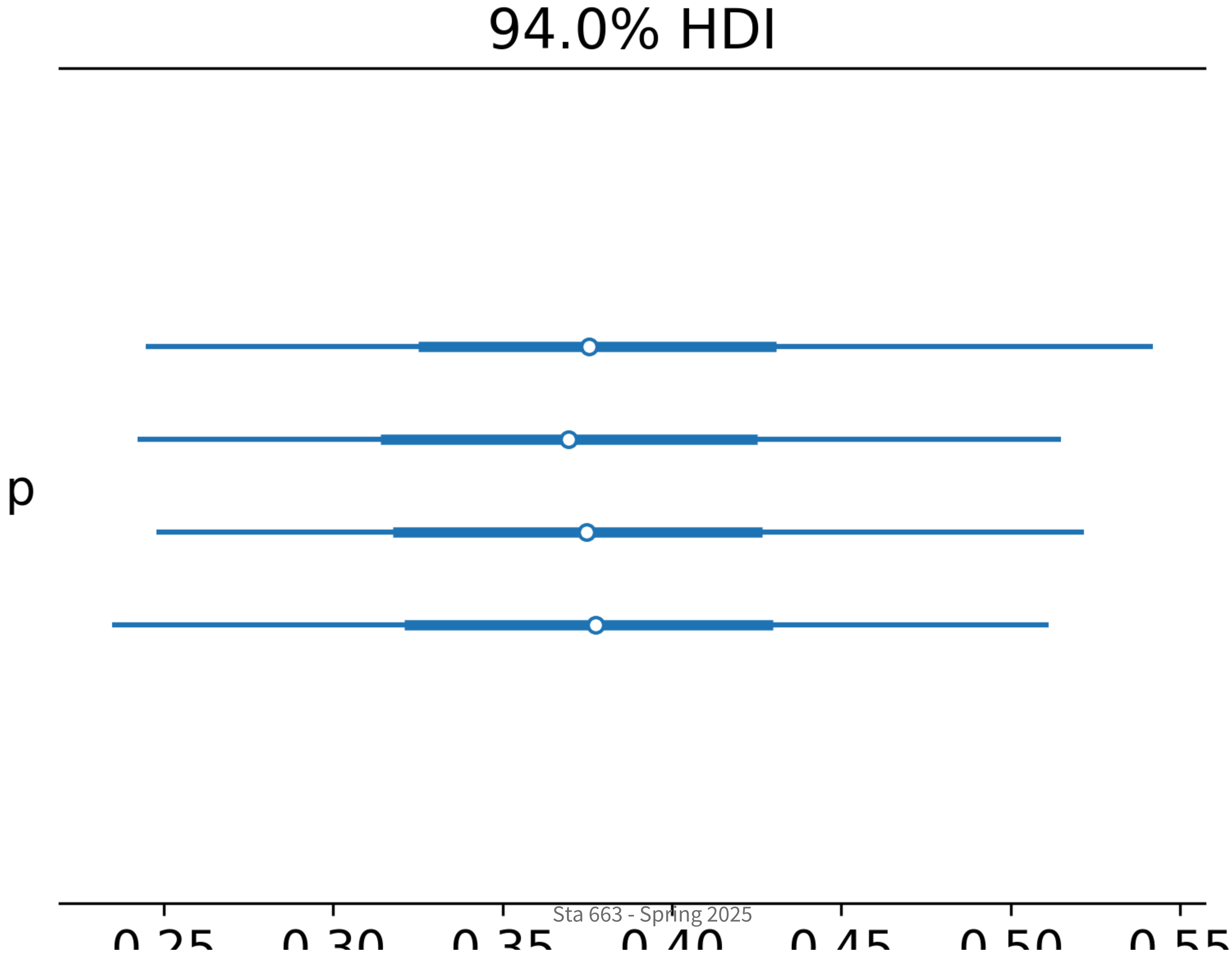
Autocorrelation plots

```
az.plot_autocorr(trace, grid=(2,2), max_lag=20)
plt.show()
```



Forest plots

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



94.0% HDI
p
0.25
0.30
0.35
0.40
0.45
0.50
0.55

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	0.376	0.077	0.236	0.52	0.002	0.001	1757.0	2546.0	1.0

individual methods are available for each statistics,

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

```
<xarray.Dataset> Size: 8B
Dimensions:  ()
Data variables:
    p        float64 8B 1.757e+03
```

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

```
<xarray.Dataset> Size: 8B
Dimensions:  ()
Data variables:
    p        float64 8B 1.0
```

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

```
<xarray.Dataset> Size: 8B
Dimensions:  ()
Data variables:
    p        float64 8B 2.546e+03
```

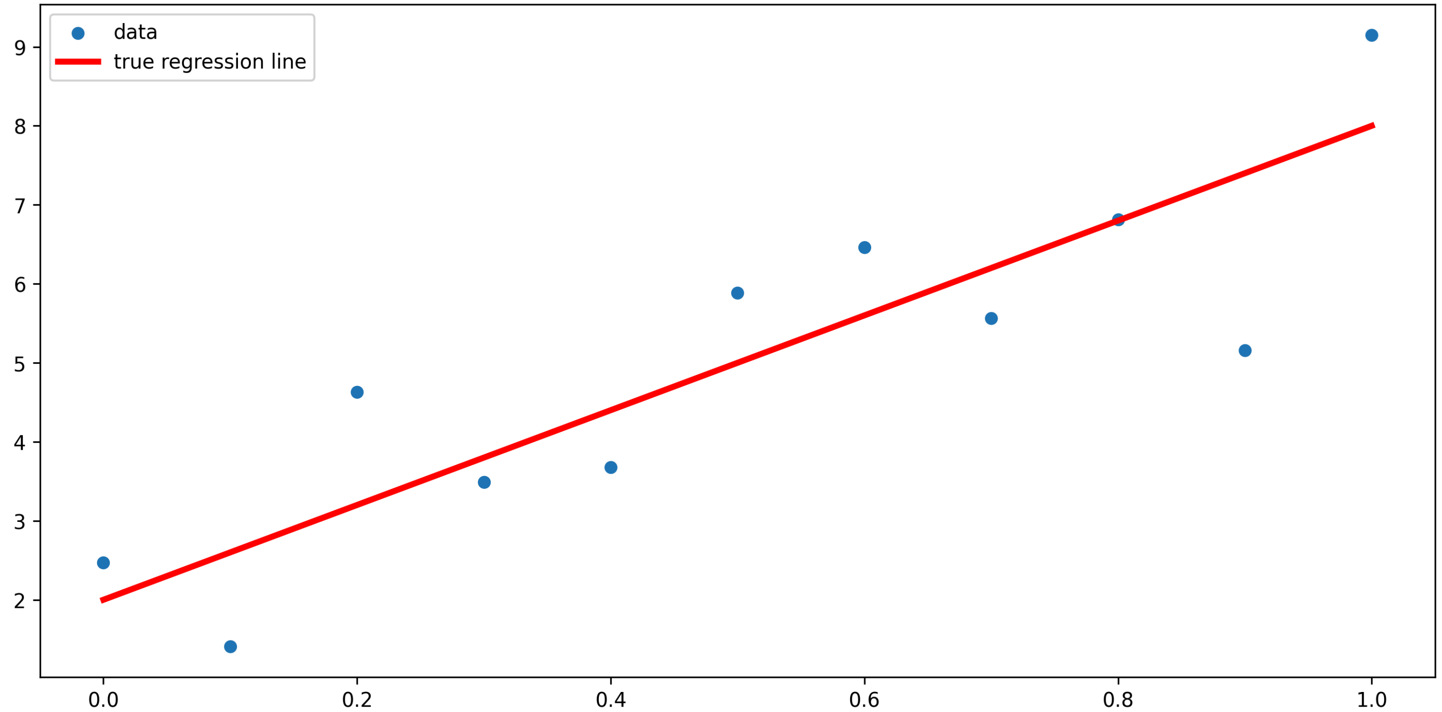
```python
print(az.mcse(trace))
```

```
<xarray.Dataset> Size: 8B
Dimensions:  ()
Data variables:
    p        float64 8B 0.001843
```

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)

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

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

More on `pm.sample()` arguments next time, but by default PyMC tunes / burns-in for 1000 iterations and then samples

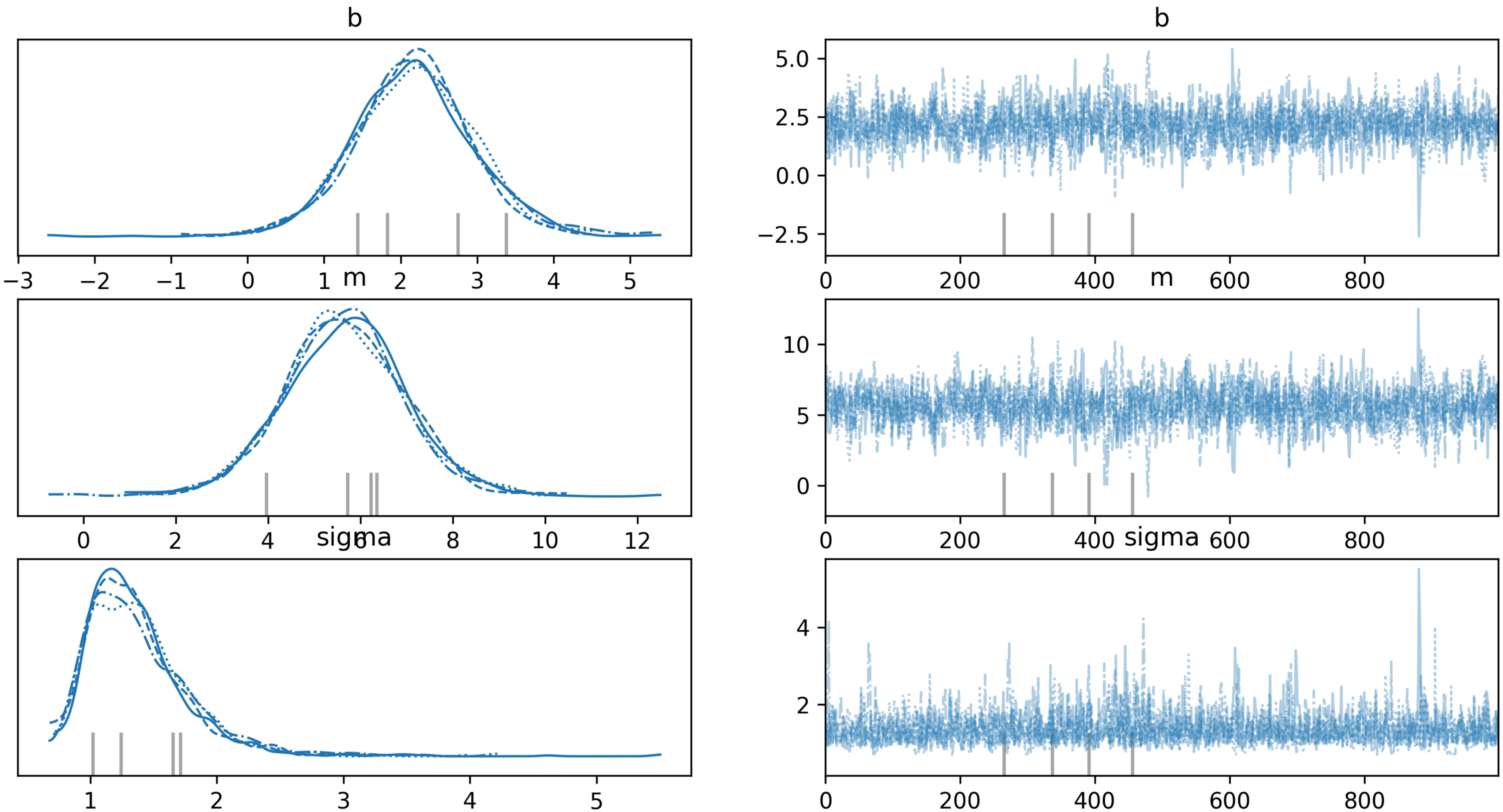
Posterior summary

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b	2.166	0.778	0.685	3.596	0.022	0.019	1230.0	1576.0	1.0
m	5.627	1.321	3.179	8.103	0.037	0.032	1282.0	1596.0	1.0
sigma	1.374	0.406	0.721	2.041	0.010	0.016	1614.0	1260.0	1.0

Trace plots

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

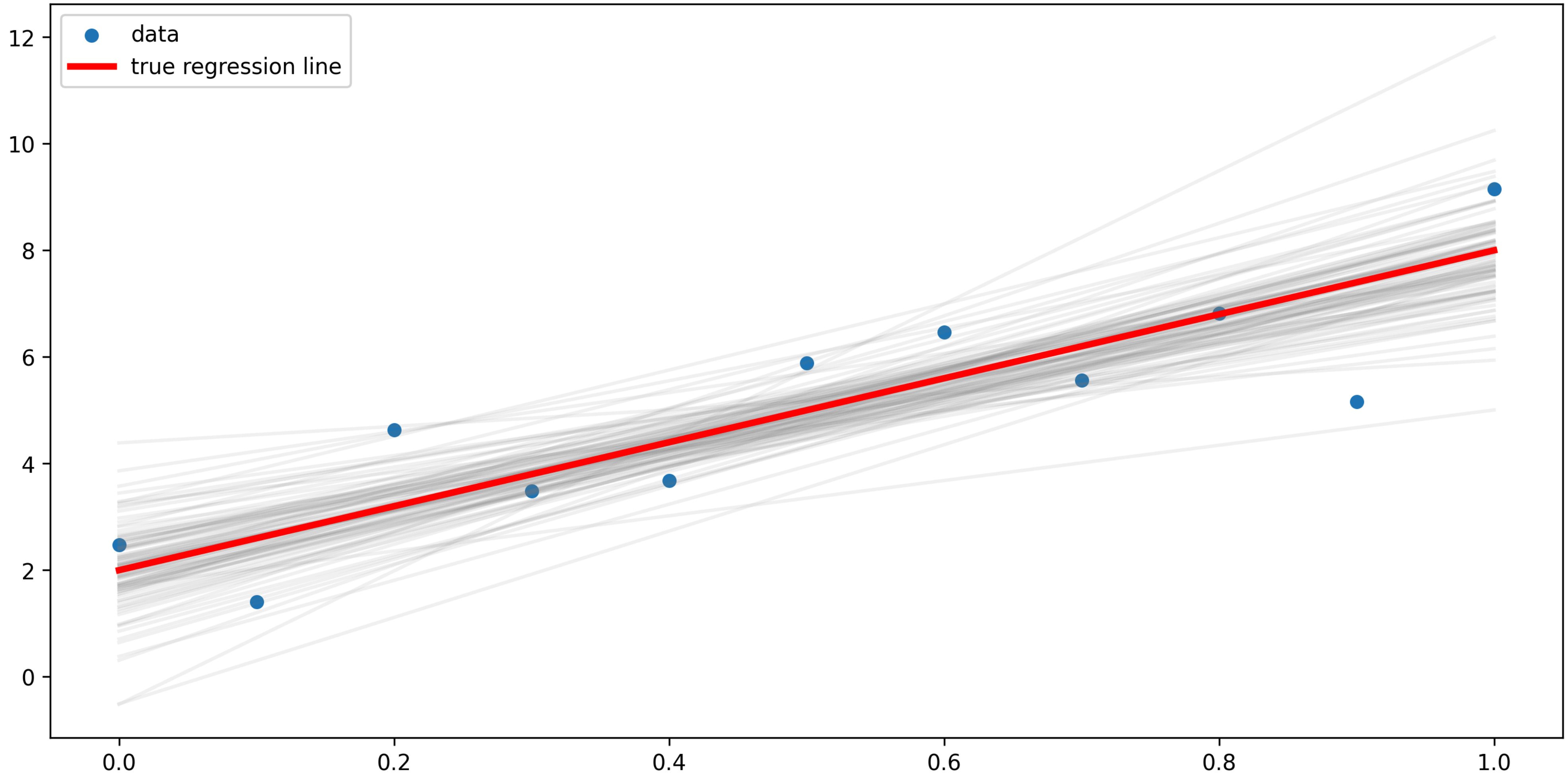


Regression line posterior draws

```python
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()
```

Regression line posterior draws



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)
```

```
pp
```

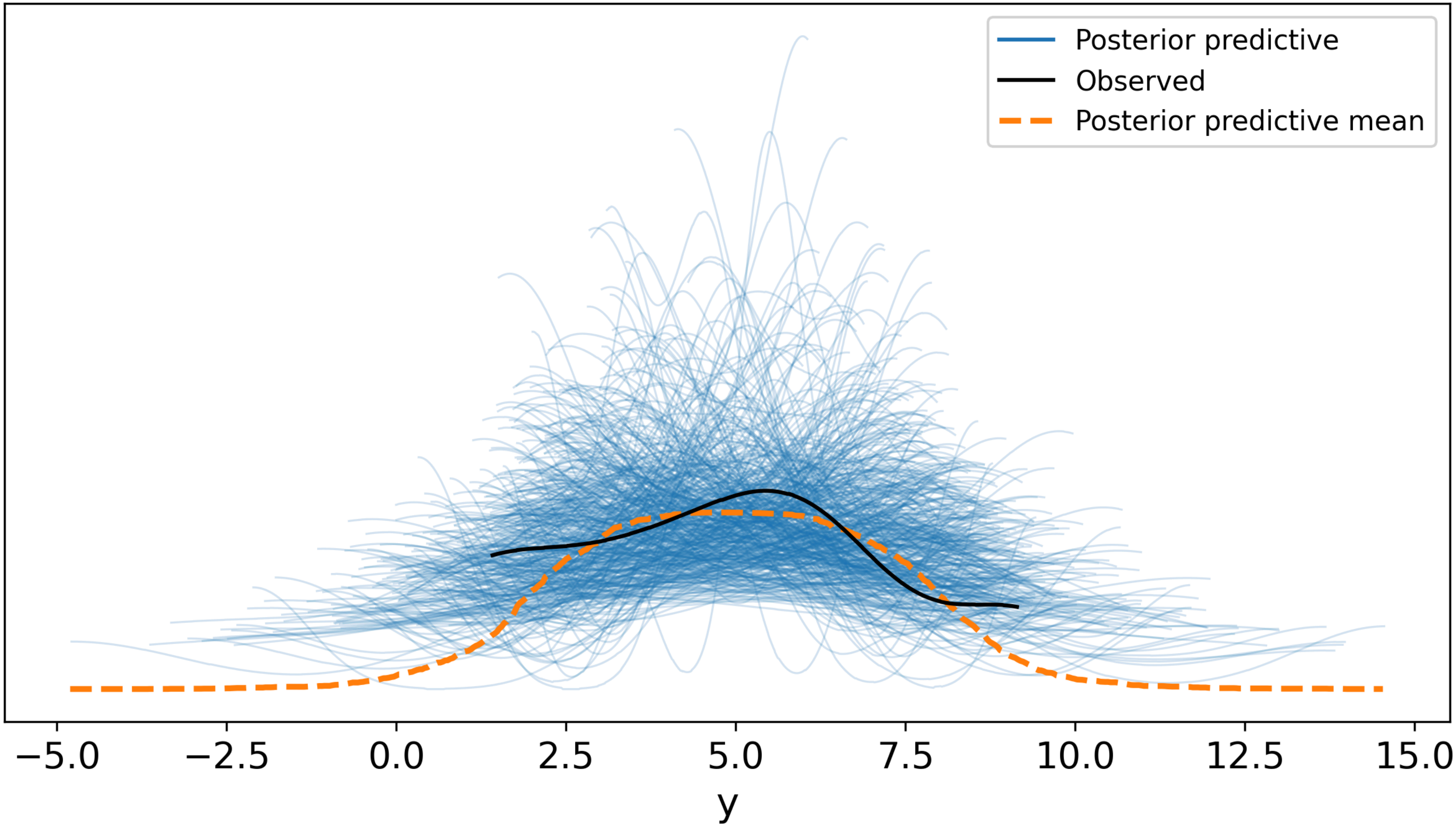
arviz.InferenceData

▸ posterior_predictive

▸ observed_data

Plotting the posterior predictive distribution

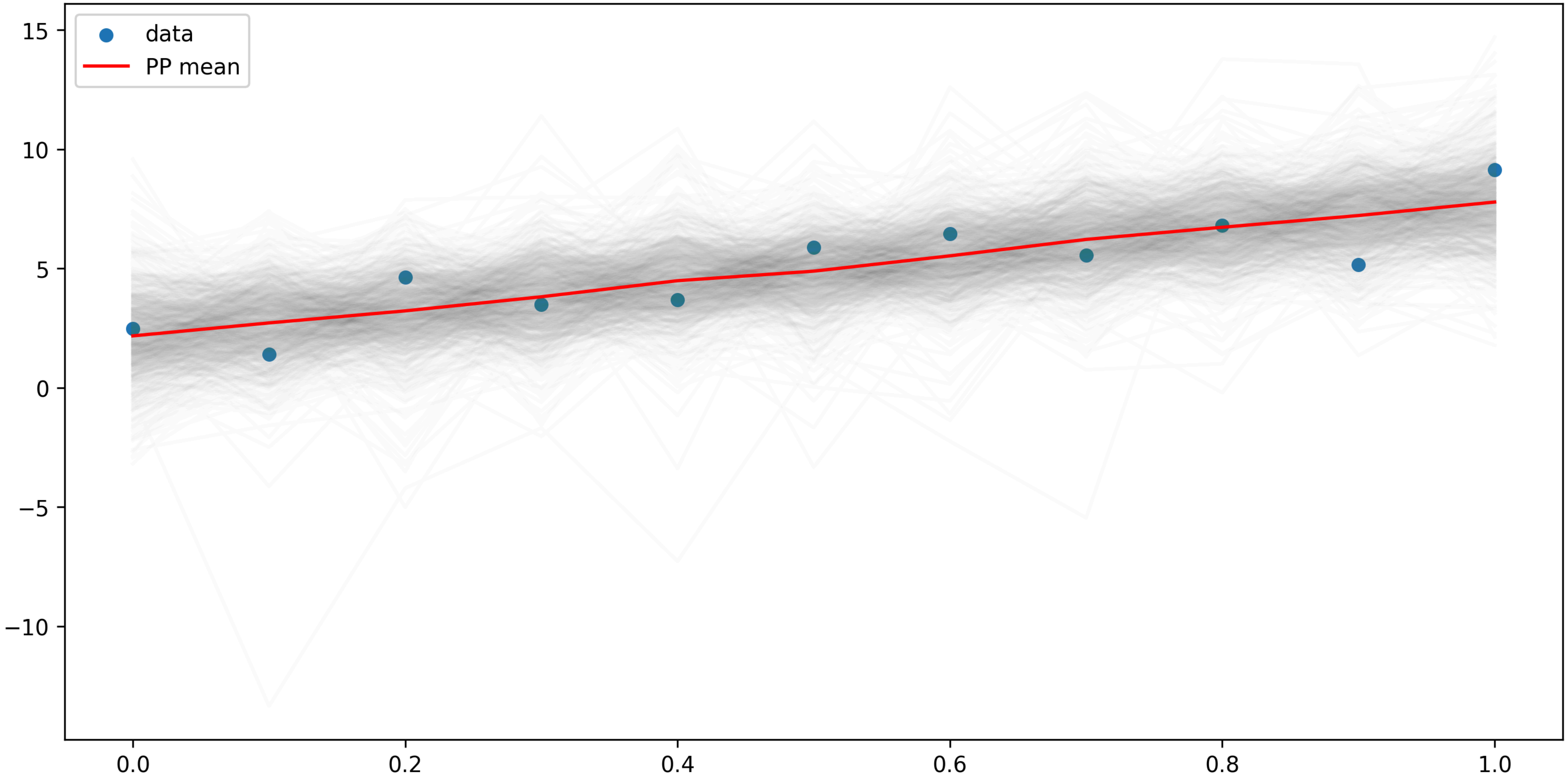
```
az.plot_ppc(pp, num_pp_samples=500)
plt.show()
```



Posterior predictive
Observed
Posterior predictive mean
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
y

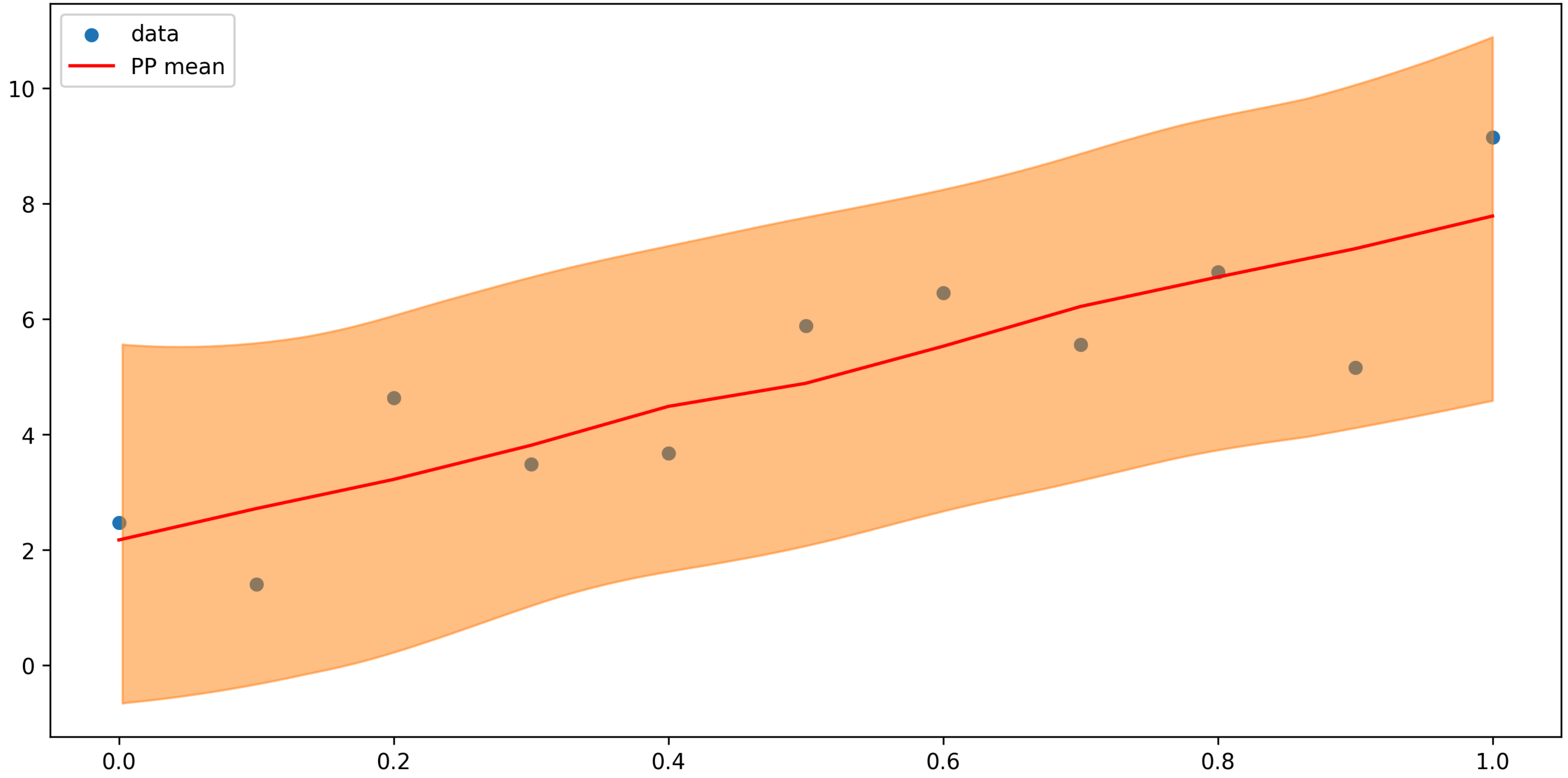
PP draws

```
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
plt.legend()
plt.show()
```


data
PP mean
15
10
5
0
−5
−10
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
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)

  likelihood = 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=Fals
```

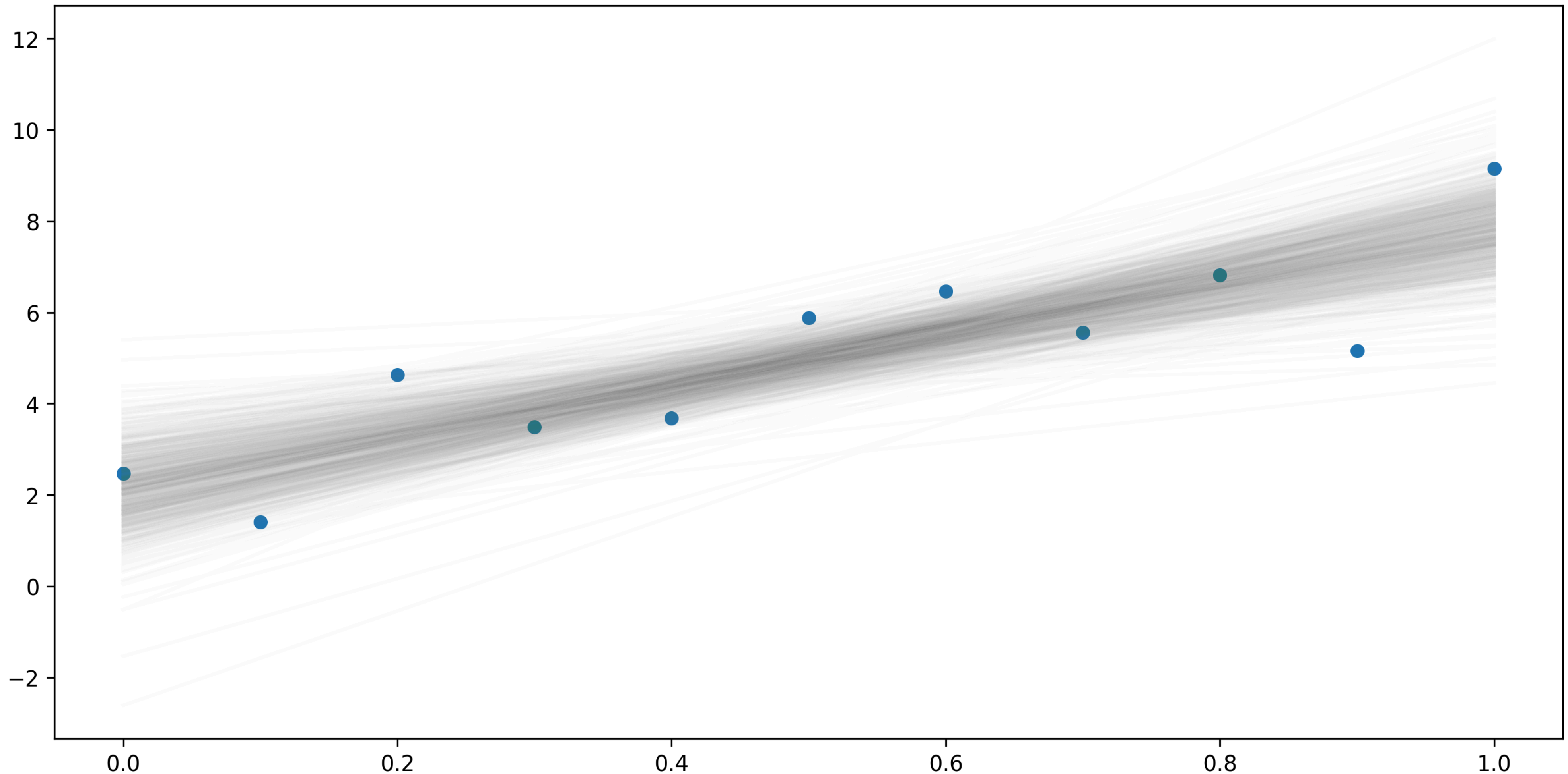
$\hat{y}$ - PP

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b	2.166	0.778	0.685	3.596	0.022	0.019	1230.0	1576.0	1.0
m	5.627	1.321	3.179	8.103	0.037	0.032	1282.0	1596.0	1.0
sigma	1.374	0.406	0.721	2.041	0.010	0.016	1614.0	1260.0	1.0
y_hat[0]	2.166	0.778	0.685	3.596	0.022	0.019	1230.0	1576.0	1.0
y_hat[1]	2.729	0.672	1.476	4.006	0.019	0.016	1294.0	1537.0	1.0
y_hat[2]	3.291	0.576	2.240	4.433	0.016	0.013	1420.0	1648.0	1.0
y_hat[3]	3.854	0.497	2.936	4.811	0.012	0.011	1715.0	1959.0	1.0
y_hat[4]	4.417	0.444	3.556	5.218	0.009	0.009	2448.0	2475.0	1.0
y_hat[5]	4.980	0.427	4.176	5.762	0.007	0.009	3762.0	2262.0	1.0
y_hat[6]	5.542	0.450	4.717	6.393	0.007	0.010	4249.0	2635.0	1.0
y_hat[7]	6.105	0.507	5.187	7.077	0.009	0.011	3614.0	2566.0	1.0
y_hat[8]	6.668	0.589	5.499	7.713	0.011	0.012	2804.0	2318.0	1.0
y_hat[9]	7.230	0.687	5.911	8.473	0.014	0.014	2358.0	2265.0	1.0
y_hat[10]	7.793	0.794	6.226	9.198	0.018	0.016	2088.0	2214.0	1.0

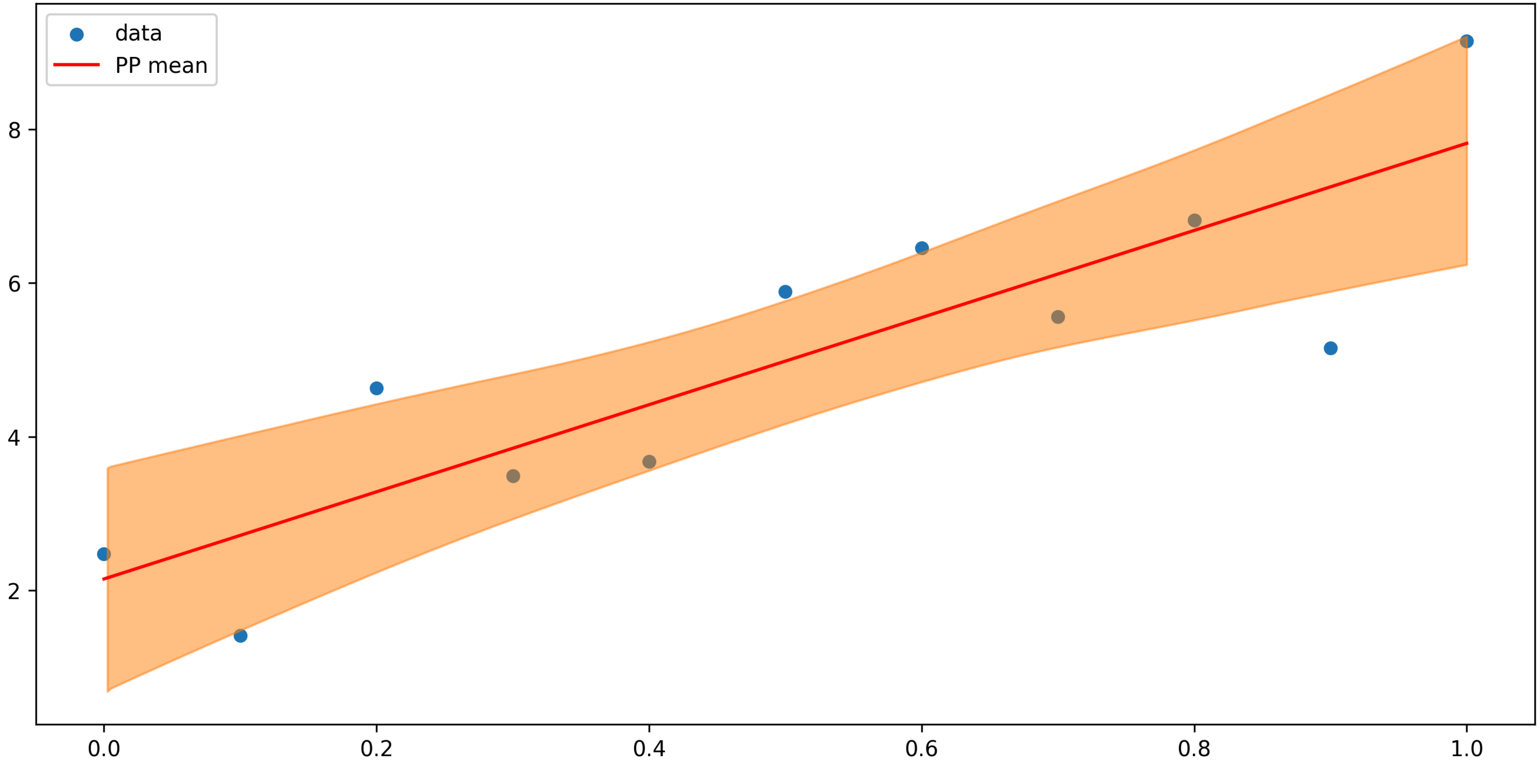
$\hat{y}$ - PP draws

```
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()
```



$\hat{y}$ - PP HDI

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



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

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)
```

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)
```

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bull
beta[0]	242.625	995.218	-1000.355	2950.690	434.973	319.426	6.0
beta[1]	-574.362	656.294	-1402.057	358.625	298.857	116.629	6.0
beta[2]	-1014.882	528.442	-1886.908	167.967	168.015	136.215	9.0
beta[3]	78.346	788.822	-860.682	1472.876	370.525	171.538	5.0
beta[4]	551.583	777.056	-454.753	1975.072	332.438	175.438	6.0
...	...	...	...	...	...	...	...
beta[96]	-702.573	880.101	-2461.944	820.584	409.925	269.151	5.0
beta[97]	117.851	813.313	-1336.065	1519.139	387.280	201.783	5.0
beta[98]	-297.373	995.011	-2420.511	1270.081	430.002	236.156	5.0
beta[99]	493.648	1248.412	-956.249	2742.285	598.336	261.016	5.0
sigma	2.498	1.021	1.298	4.470	0.363	0.063	9.0

101 rows × 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)
```

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

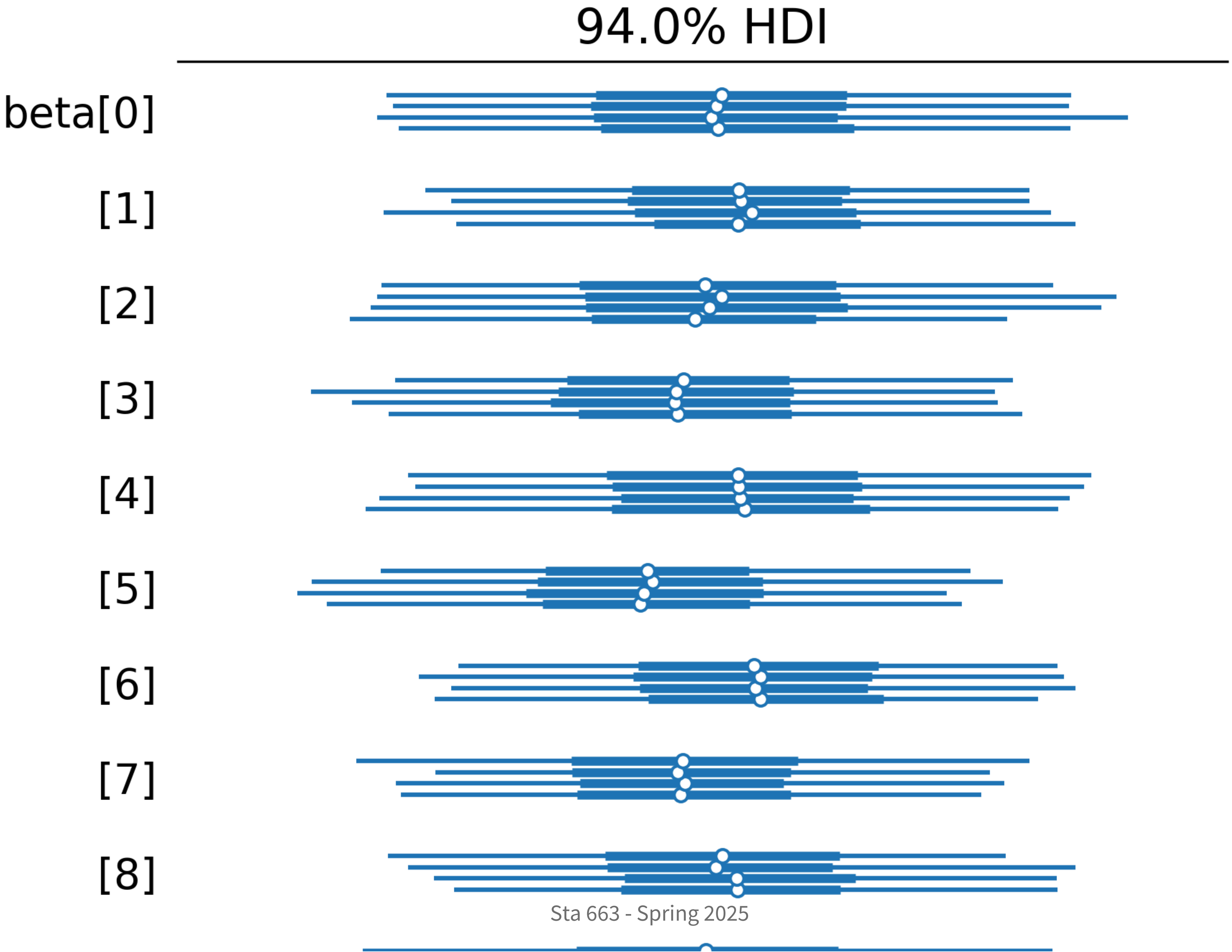
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_ta
beta[0]	0.311	7.613	-13.351	15.058	0.105	0.137	5329.0	2526.(
beta[1]	1.233	6.587	-11.725	13.485	0.087	0.115	5716.0	2590.(
beta[2]	-0.120	7.743	-14.607	13.999	0.104	0.175	5597.0	2117.(
beta[3]	-1.485	6.990	-14.770	11.764	0.089	0.125	6156.0	2846.(
beta[4]	0.877	7.454	-13.828	14.448	0.103	0.125	5269.0	2631.(
...	...	...	...	...	...	...	...	...
beta[96]	-0.377	6.414	-12.167	11.563	0.089	0.107	5098.0	2949.(
beta[97]	-0.783	6.470	-13.404	10.882	0.089	0.109	5318.0	2669.(
beta[98]	1.443	6.720	-10.492	14.634	0.105	0.120	4091.0	2324.(
beta[99]	-1.201	6.854	-13.770	12.981	0.093	0.121	5426.0	2993.(
sigma	2.259	1.056	0.635	4.159	0.117	0.055	75.0	143.0

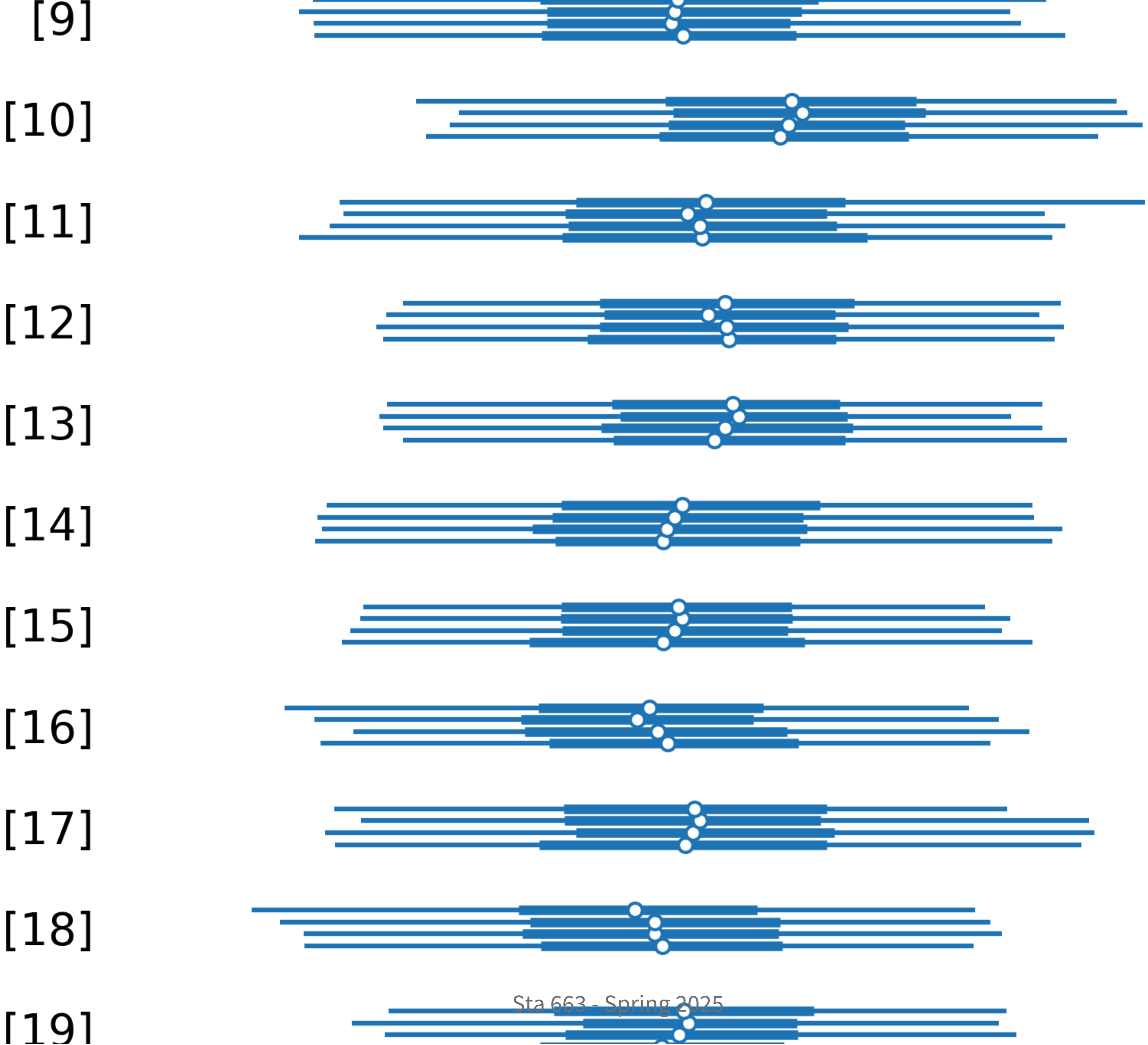
101 rows × 9 columns

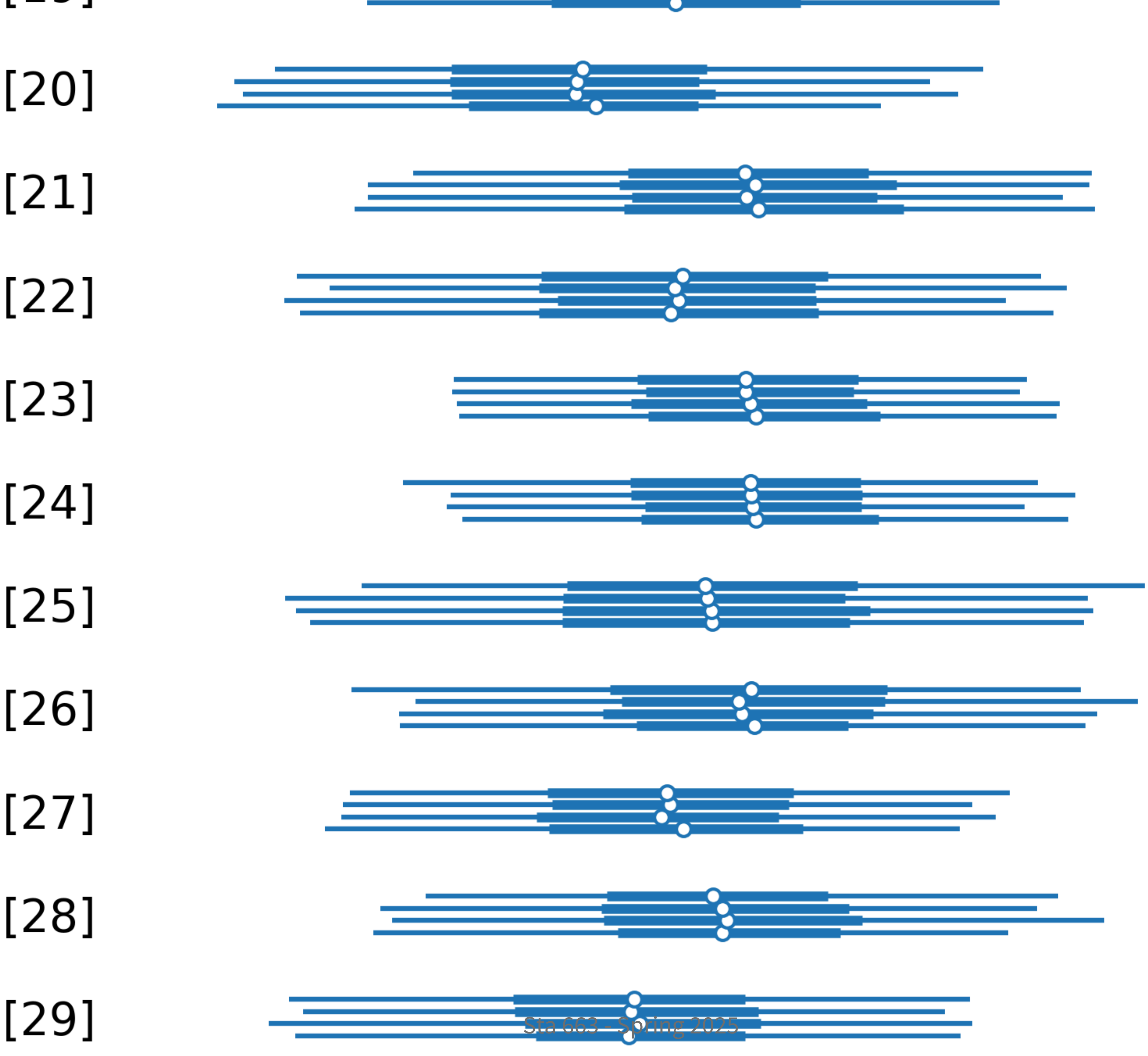
```python
az.summary(trace).iloc[[10,20,30,40,50,60,70,80]]
```

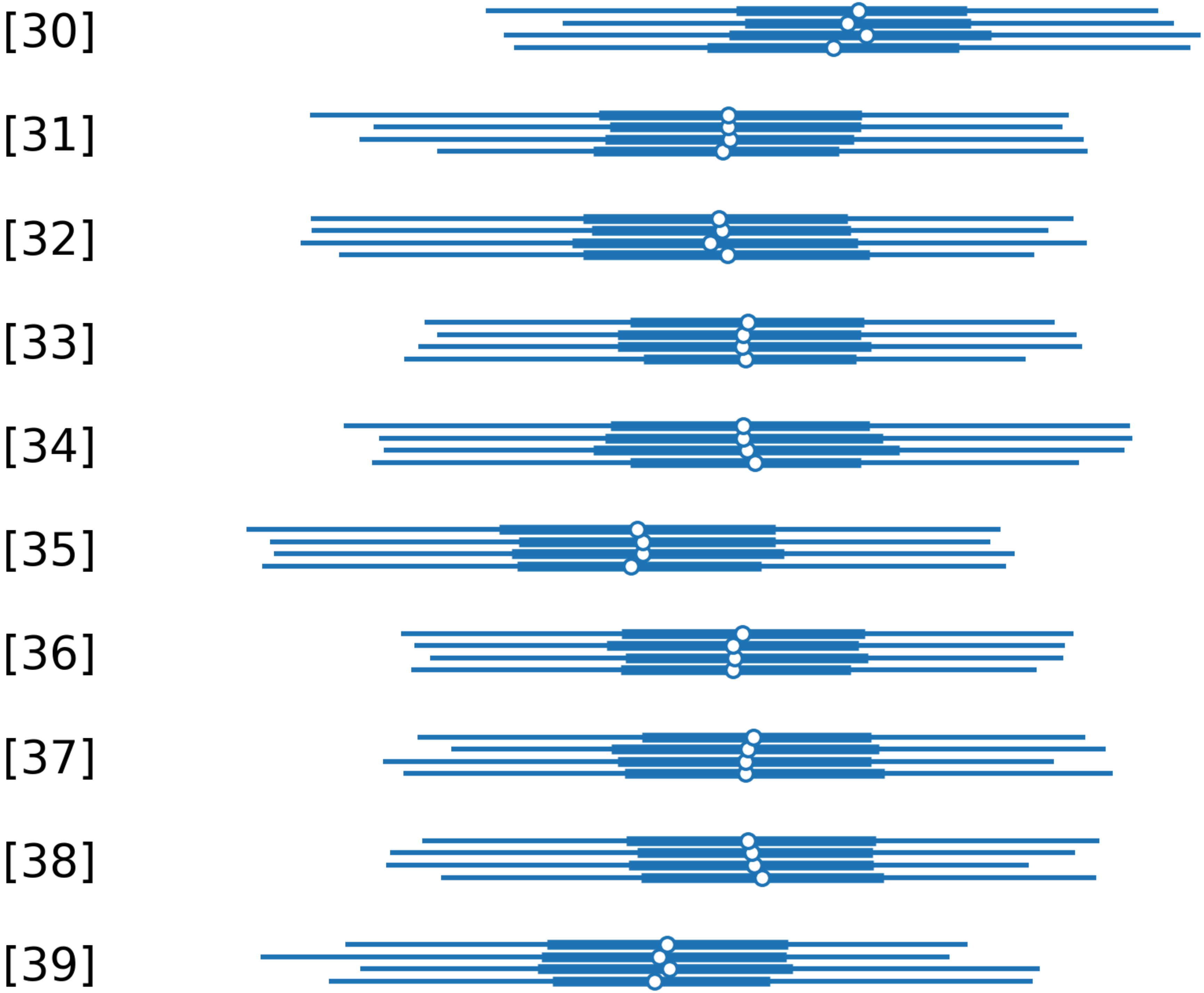
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_ta
beta[10]	3.951	7.004	-9.375	16.974	0.115	0.127	3729.0	2376.(
beta[20]	-4.504	7.036	-18.159	8.207	0.097	0.115	5239.0	2944.(
beta[30]	5.473	6.723	-7.496	17.680	0.104	0.126	4182.0	2049.(
beta[40]	-4.949	8.112	-19.896	11.253	0.097	0.162	7039.0	2573.(
beta[50]	5.392	6.492	-6.410	17.520	0.089	0.106	5294.0	3012.(
beta[60]	-5.692	7.026	-18.329	8.414	0.097	0.128	5259.0	2191.(
beta[70]	4.644	7.427	-8.624	19.128	0.110	0.146	4585.0	1979.(
beta[80]	-7.800	6.097	-19.094	3.737	0.082	0.095	5551.0	3325.(

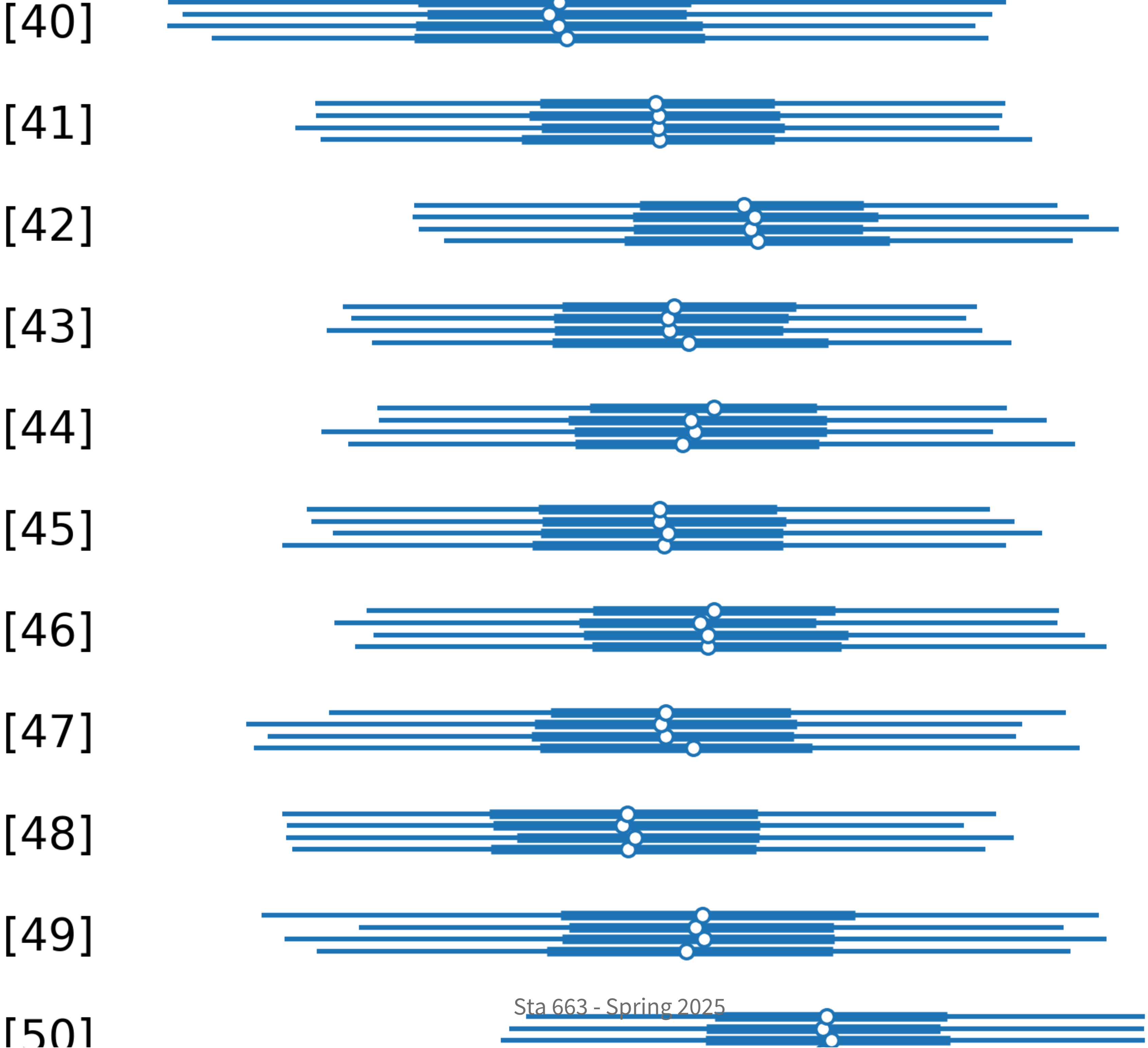
```
ax = az.plot_forest(trace)
plt.tight_layout()
plt.show()
```

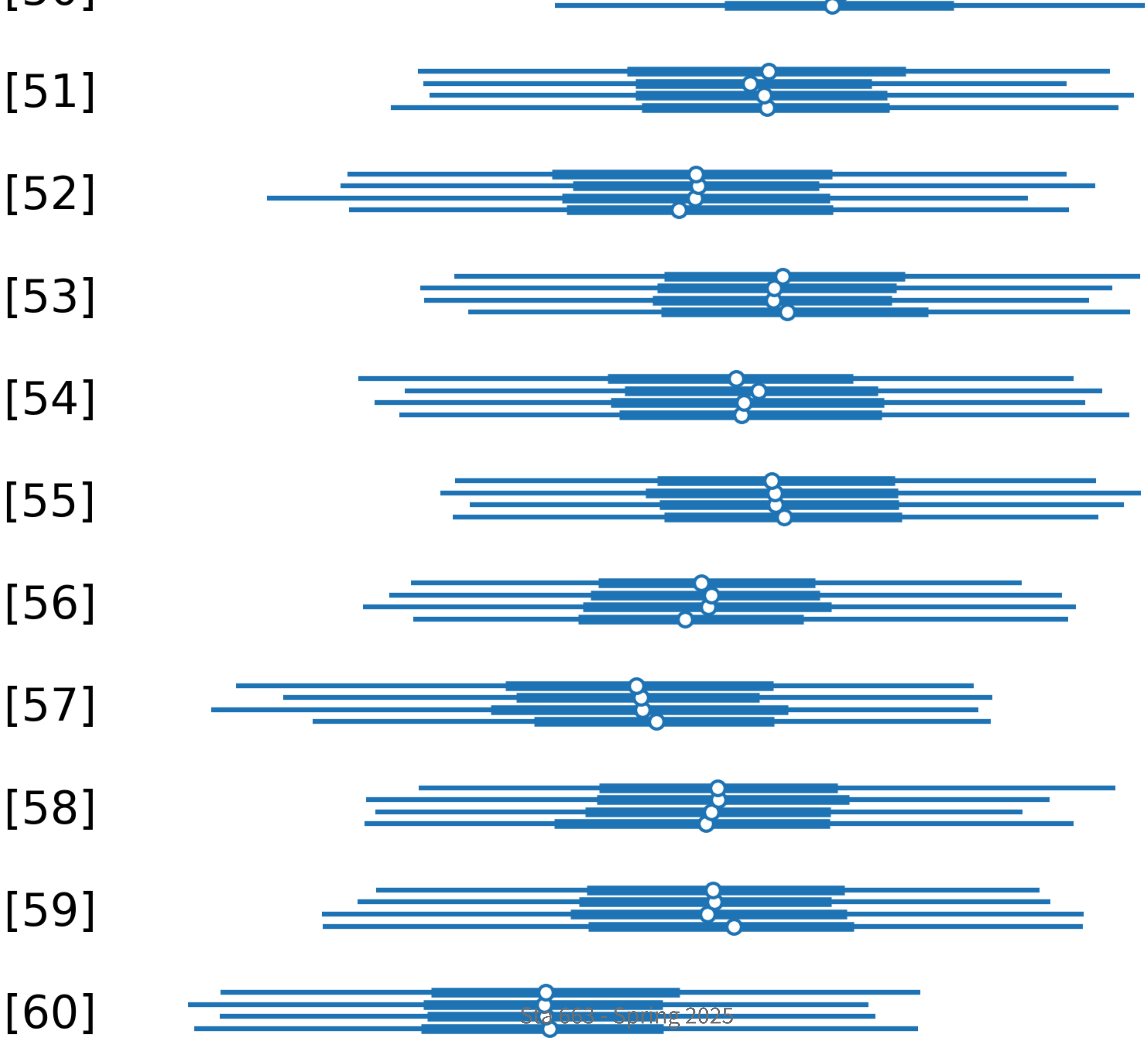

94.0% HDI
beta[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]

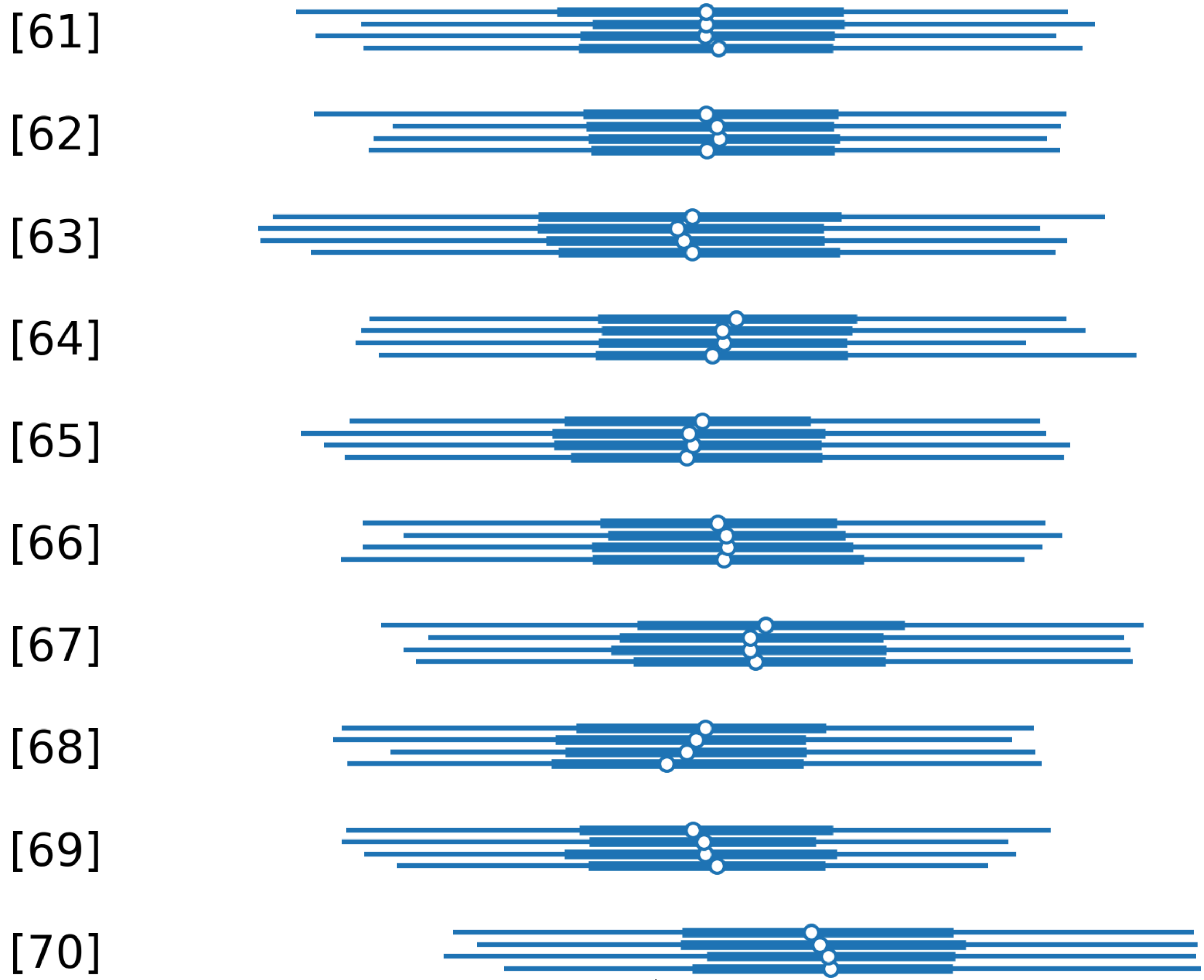

[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]

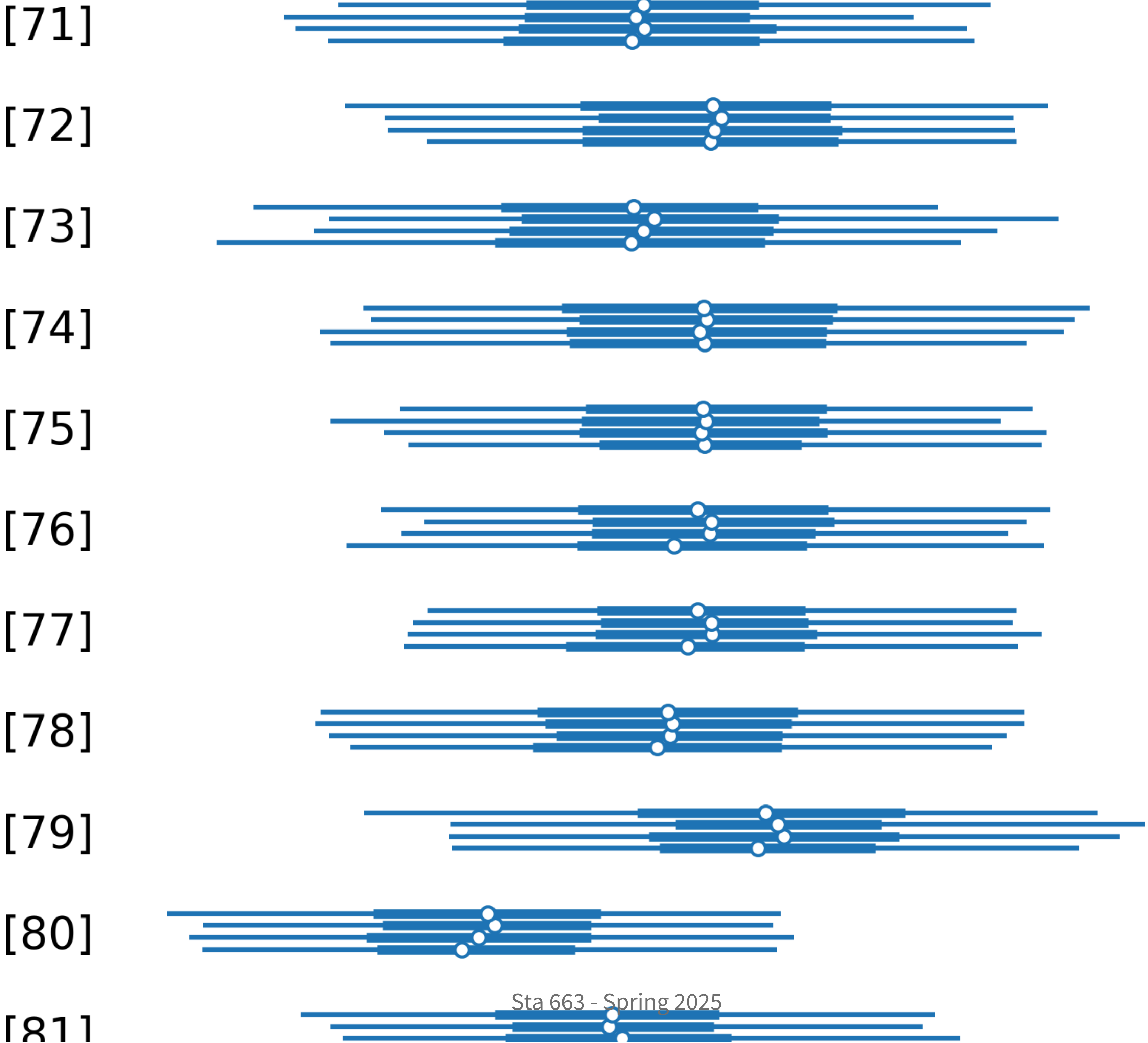

[20]
[21]
[22]
[23]
[24]
[25]
[26]
[27]
[28]
[29]

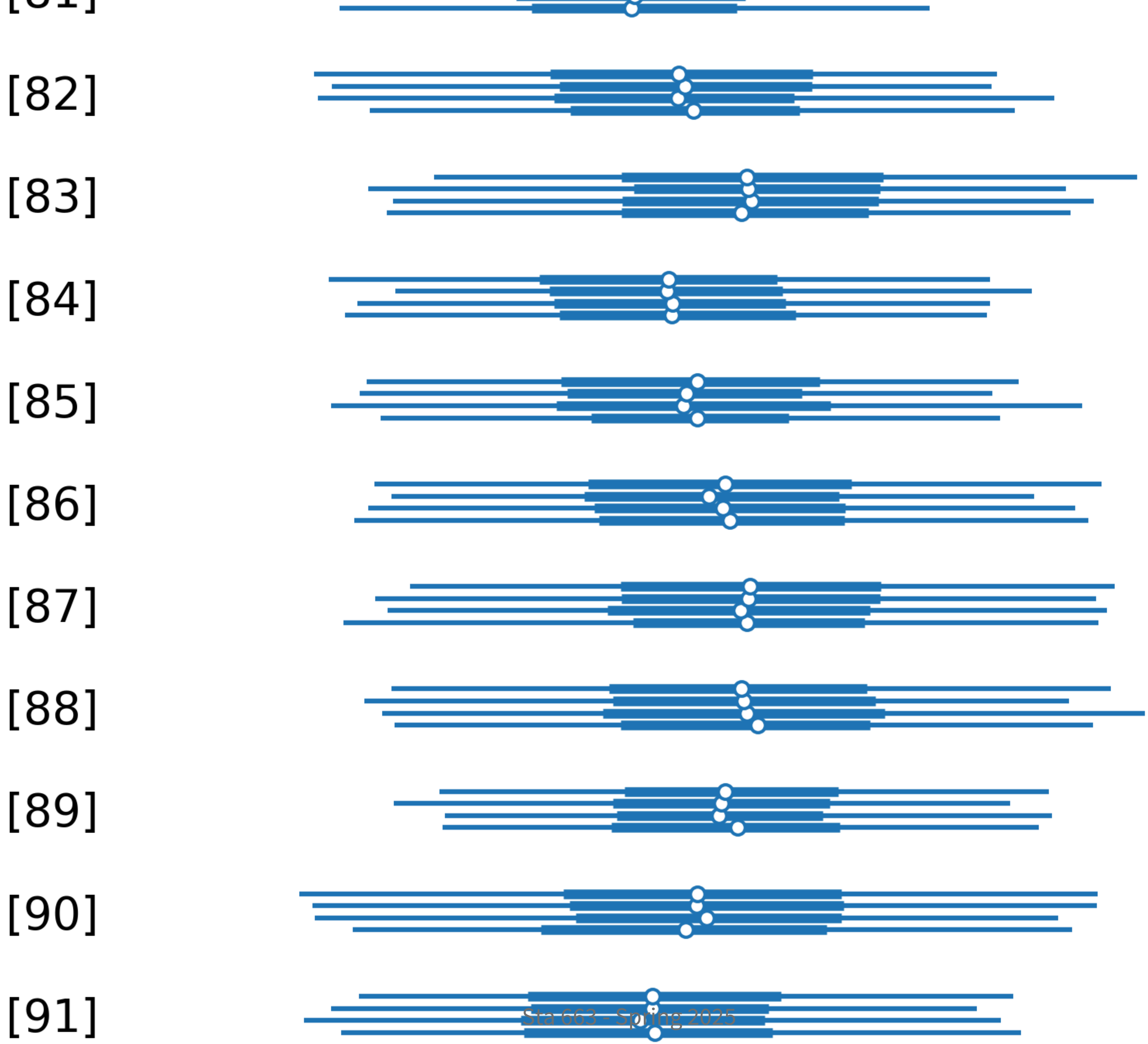

[30]
[31]
[32]
[33]
[34]
[35]
[36]
[37]
[38]
[39]

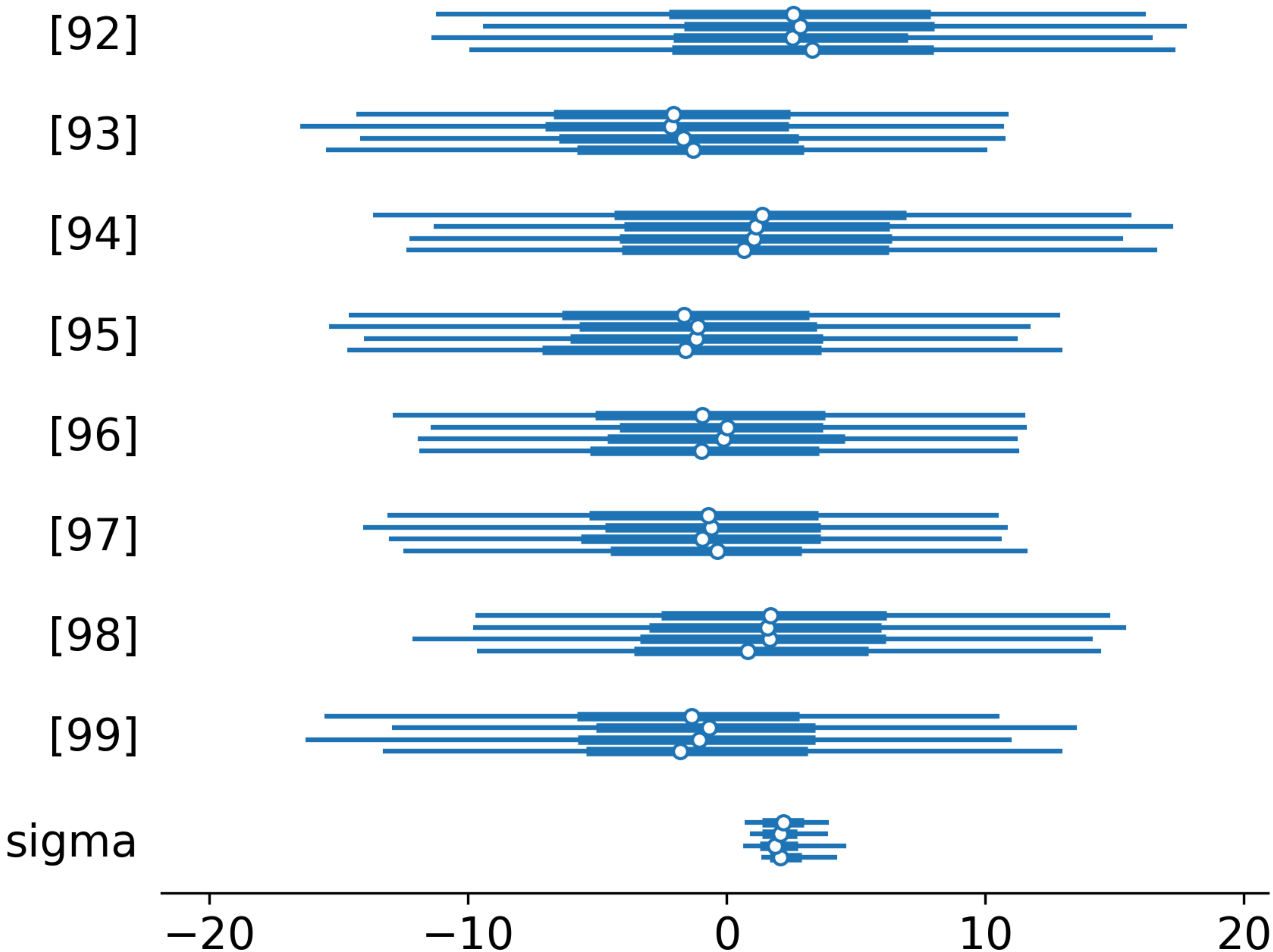

[40]
[41]
[42]
[43]
[44]
[45]
[46]
[47]
[48]
[49]
[50]


[50]
[51]
[52]
[53]
[54]
[55]
[56]
[57]
[58]
[59]
[60]


[61]
[62]
[63]
[64]
[65]
[66]
[67]
[68]
[69]
[70]


[71]
[72]
[73]
[74]
[75]
[76]
[77]
[78]
[79]
[80]
[81]


[82]
[83]
[84]
[85]
[86]
[87]
[88]
[89]
[90]
[91]

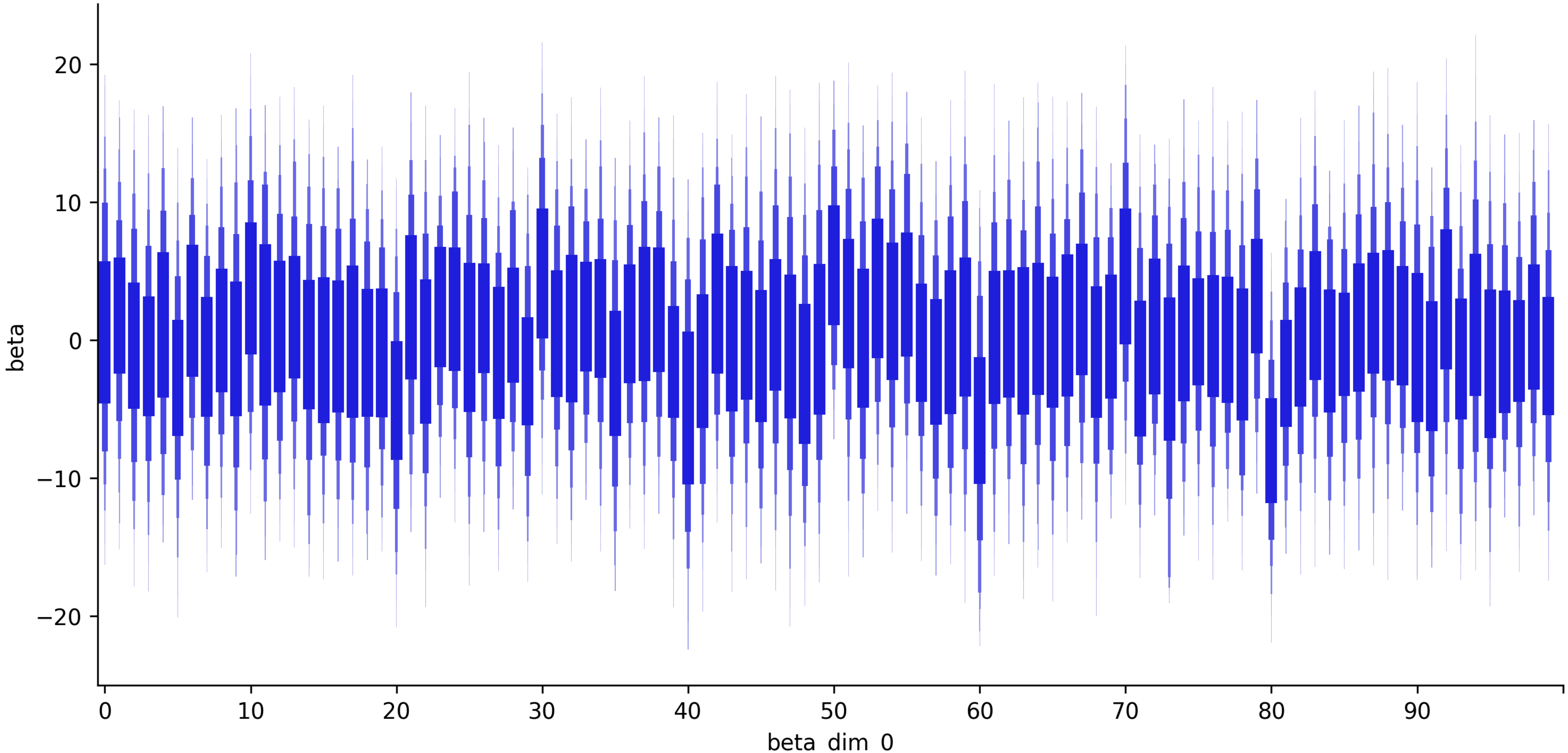

[92]
[93]
[94]
[95]
[96]
[97]
[98]
[99]
sigma
−20
−10
0
10
20

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, colo
  plt.tight_layout()
  plt.xticks(range(0,110,10))
  plt.show()
```

```
plot_slope(trace)
```



Laplace Prior

```python
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)
```

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

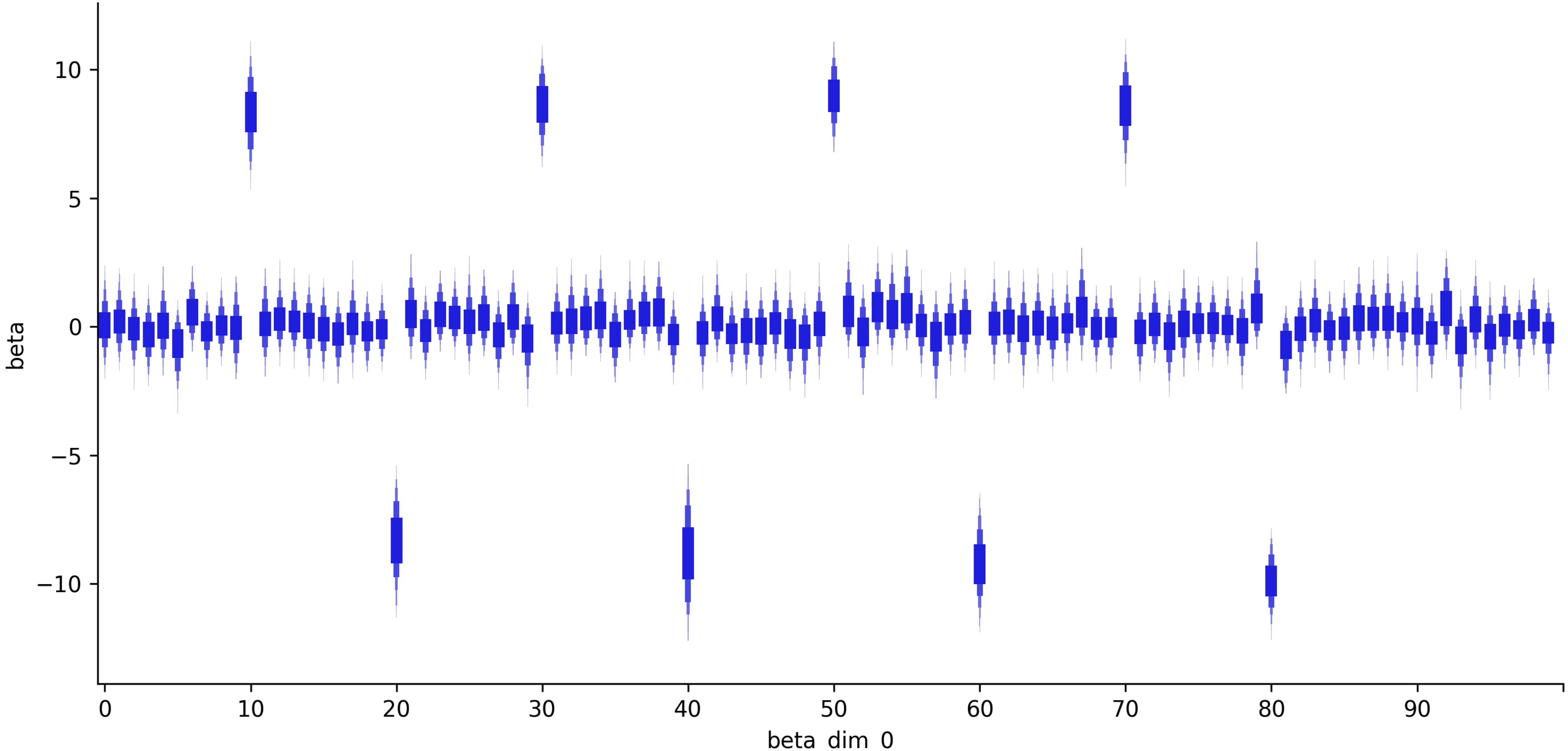
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_ta
beta[0]	0.079	0.846	-1.540	1.765	0.015	0.020	3605.0	2149.(
beta[1]	0.212	0.738	-1.148	1.709	0.011	0.016	4542.0	2608.(
beta[2]	-0.057	0.803	-1.692	1.382	0.013	0.017	3748.0	2732.(
beta[3]	-0.280	0.777	-1.826	1.146	0.014	0.017	3336.0	1972.(
beta[4]	0.075	0.859	-1.596	1.708	0.017	0.026	2979.0	1849.(
...	...	...	...	...	...	...	...	...
beta[96]	0.061	0.752	-1.435	1.558	0.017	0.018	2233.0	1258.(
beta[97]	-0.143	0.718	-1.536	1.257	0.013	0.016	2932.0	2380.(
beta[98]	0.300	0.743	-0.997	1.752	0.016	0.016	2268.0	2275.(
beta[99]	-0.289	0.775	-1.839	1.144	0.015	0.017	2792.0	1483.(
sigma	0.902	0.497	0.276	1.848	0.058	0.032	70.0	141.0

101 rows × 9 columns

```
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_ta
beta[10]	8.349	1.248	5.734	10.493	0.027	0.023	2153.0	2169.(
beta[20]	-8.315	1.294	-10.647	-5.822	0.027	0.021	2249.0	2290.(
beta[30]	8.627	1.004	6.788	10.530	0.026	0.017	1458.0	1200.(
beta[40]	-8.772	1.641	-11.756	-5.558	0.043	0.035	1556.0	1303.(
beta[50]	9.003	1.010	7.089	10.897	0.022	0.018	2027.0	2524.(
beta[60]	-9.230	1.174	-11.417	-6.972	0.034	0.027	1178.0	597.0
beta[70]	8.512	1.172	6.203	10.570	0.025	0.021	2260.0	2035.(
beta[80]	-9.919	0.869	-11.531	-8.265	0.018	0.015	2224.0	2657.(

```python
plot_slope(trace)
```



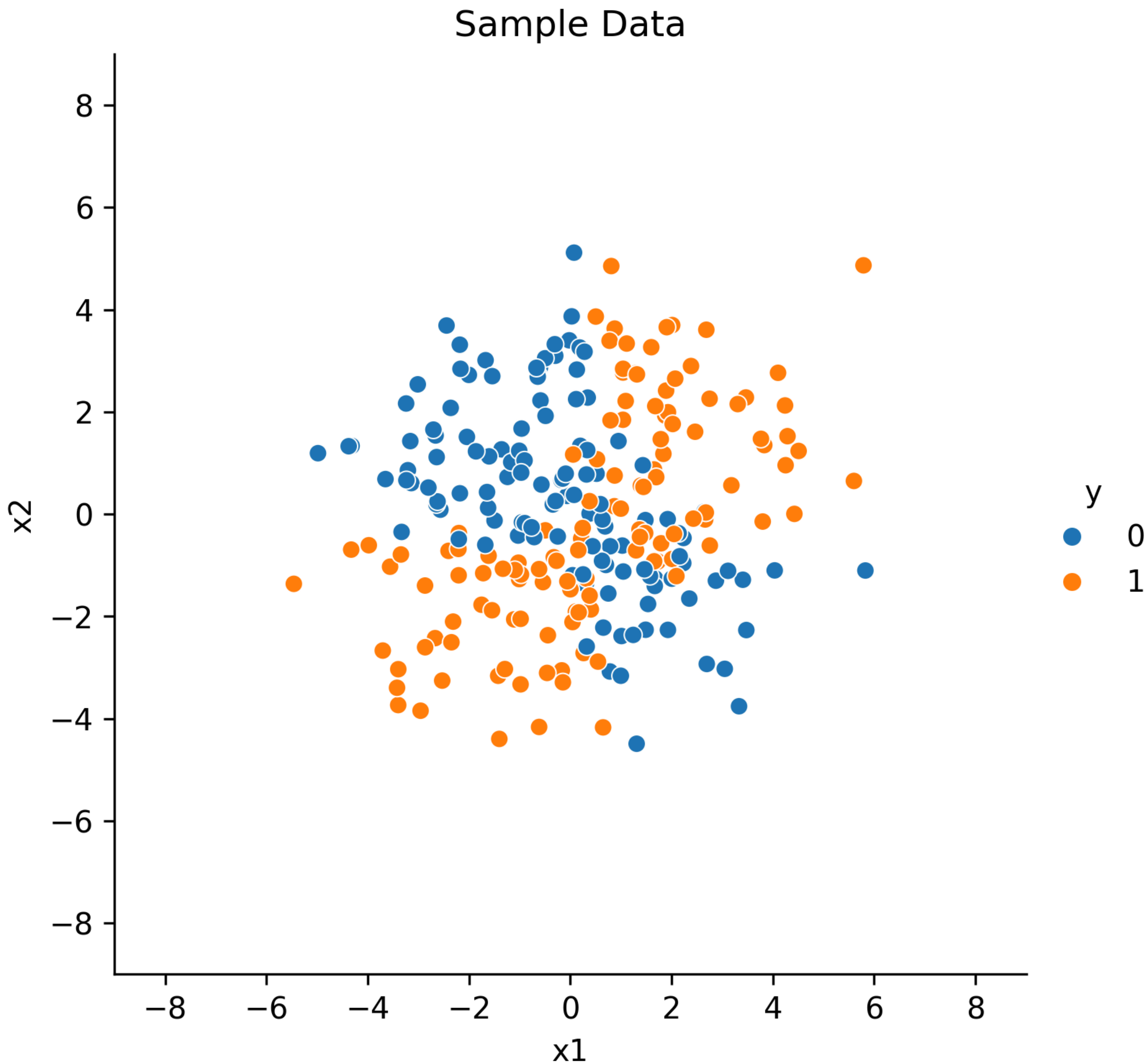
Demo 3 - Logistic Regression

Based on PyMC Out-Of-Sample Predictions example

Data

	x1	x2	y
0	-3.207674	0.859021	0
1	0.128200	2.827588	0
2	1.481783	-0.116956	0
3	0.305238	-1.378604	0
4	1.727488	-0.926357	1
...	...	...	...
245	-2.182813	3.314672	0
246	-2.362568	2.078652	0
247	0.114571	2.249021	0
248	2.093975	-1.212528	1
249	1.241667	-2.363412	0

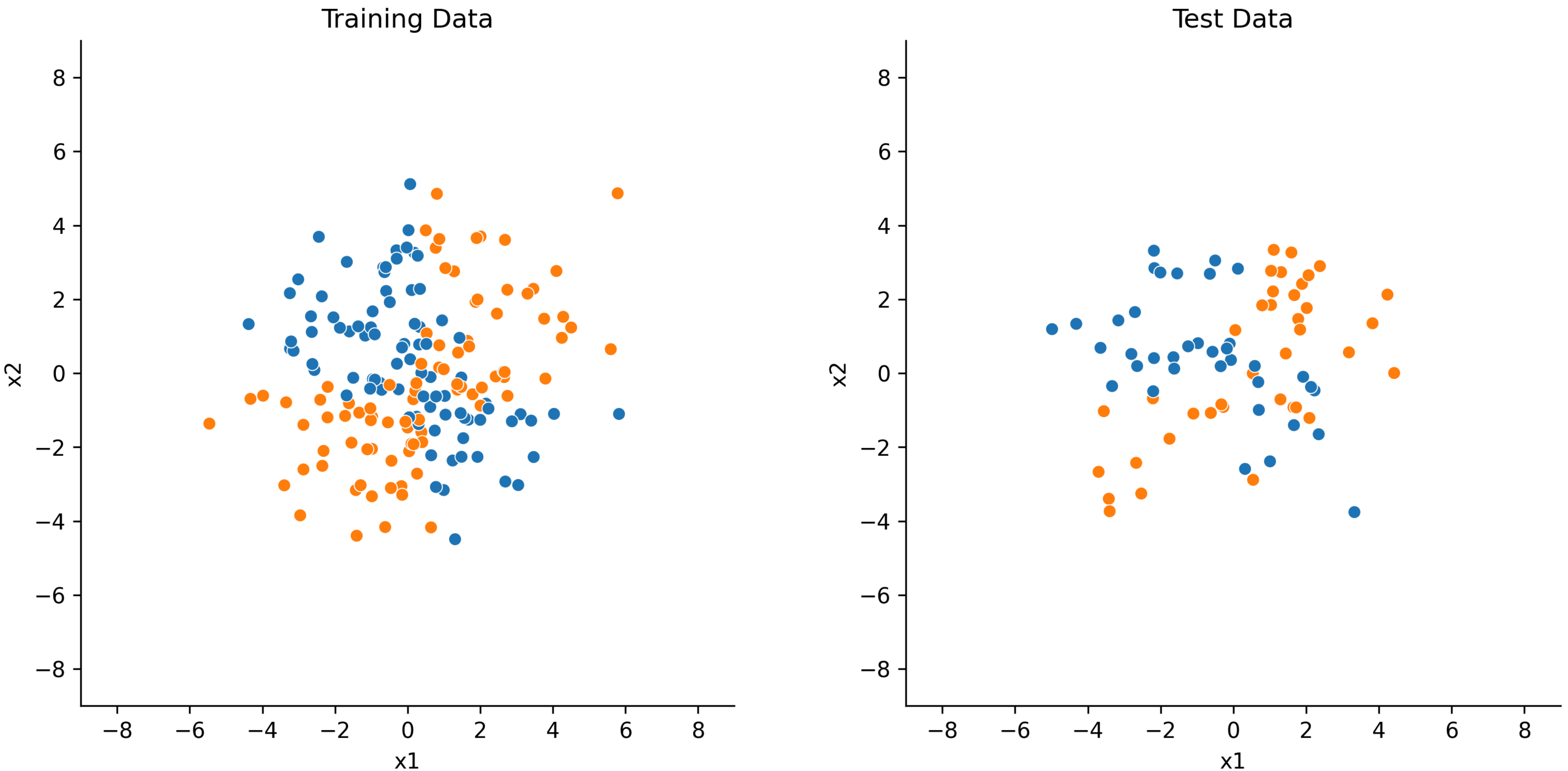
250 rows × 3 columns



Test-train split

```
from sklearn.model_selection import train_test_split

y, X = patsy.dmatrices("y ~ x1 * x2", data=df)
X_lab = X.design_info.column_names
y_lab = y.design_info.column_names
y = np.asarray(y).flatten()
X = np.asarray(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7)
```


Training Data
Test Data
x1
x2
−8
−6
−4
−2
0
2
4
6
8

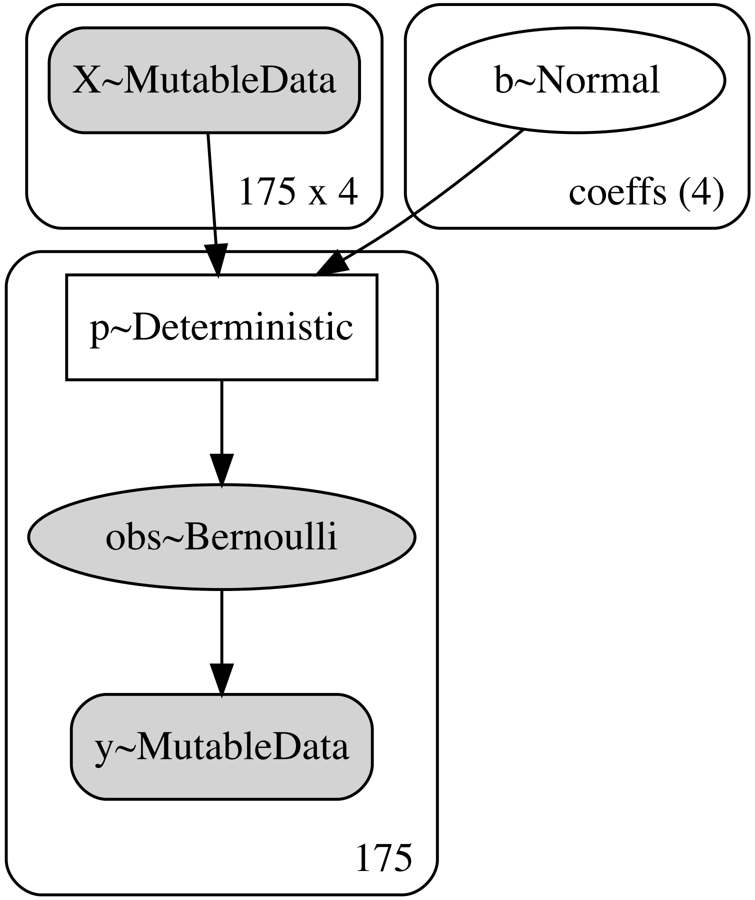
Model

```python
with pm.Model(coords = {"coeffs": X_lab}) as model:
    # data containers
    X = pm.MutableData("X", X_train)
    y = pm.MutableData("y", y_train)
    # priors
    b = pm.Normal("b", mu=0, sigma=3, dims="coeffs")
    # linear model
    mu = X @ b
    # link function
    p = pm.Deterministic("p", pm.math.invlogit(mu))
    # likelihood
    obs = pm.Bernoulli("obs", p=p, observed=y)
```

Registers the value as a SharedVariable with the model - it can then be altered in value or shape (but not dimensionality)

Visualizing models

```
pm.model_to_graphviz(model)
```



Fitting

```
with model:
    post = pm.sample(progressbar=False, random_seed=1234)
```

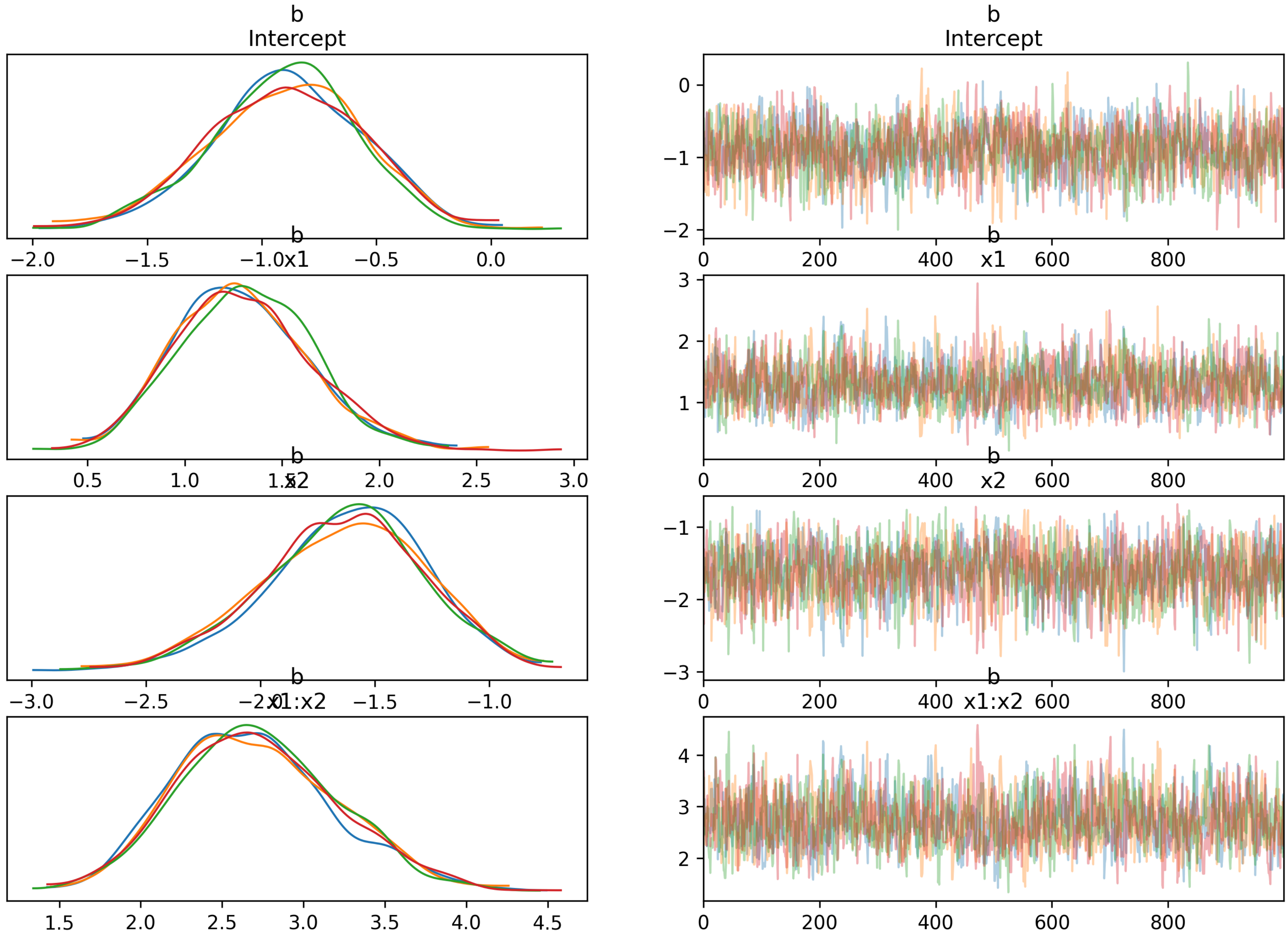
```
az.summary(post)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b[Intercept]	-0.895	0.330	-1.513	-0.294	0.010	0.005	1203.0	2281.0	1.0
b[x1]	1.299	0.341	0.662	1.913	0.010	0.006	1172.0	2033.0	1.0
b[x2]	-1.634	0.352	-2.339	-1.024	0.010	0.006	1297.0	1947.0	1.0
b[x1:x2]	2.714	0.496	1.843	3.660	0.014	0.009	1199.0	1766.0	1.0
p[0]	0.938	0.036	0.871	0.992	0.001	0.001	2382.0	2704.0	1.0
...	...	...	...	...	...	...	...	...	...
p[170]	0.000	0.000	0.000	0.000	0.000	0.000	1480.0	2089.0	1.0
p[171]	0.222	0.109	0.047	0.430	0.002	0.002	3924.0	3036.0	1.0
p[172]	1.000	0.000	1.000	1.000	0.000	0.000	1936.0	2541.0	1.0
p[173]	0.792	0.065	0.670	0.907	0.001	0.001	2632.0	2880.0	1.0
p[174]	0.623	0.075	0.480	0.760	0.001	0.001	3234.0	3044.0	1.0

179 rows × 9 columns

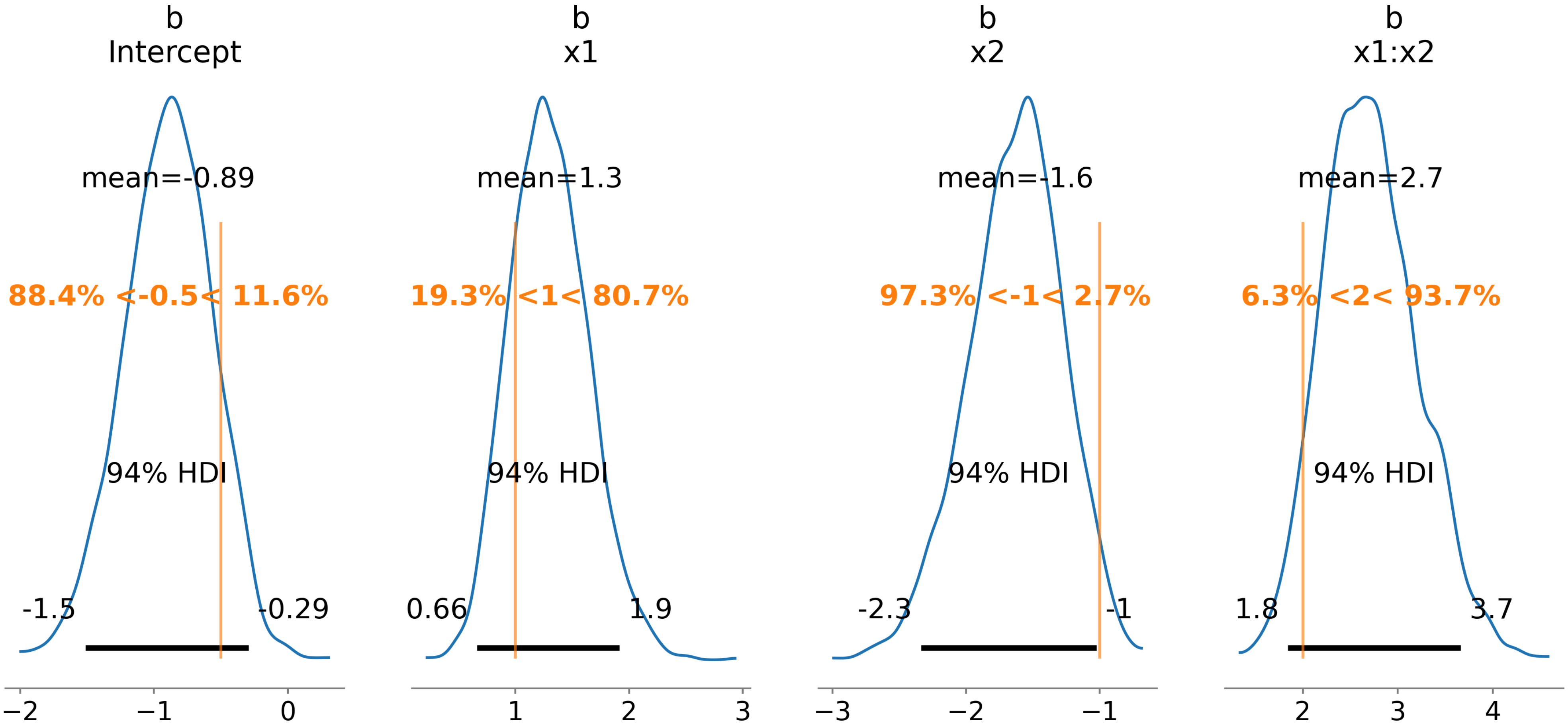
Trace plots

```
az.plot_trace(post, var_names="b", compact=False)
plt.show()
```


b
Intercept
b
x1
b
x2
b
x1:x2

Posterior plots

```
az.plot_posterior(
  post, var_names=["b"], ref_val=[intercept, beta_x1, beta_x2, beta_interaction],
  figsize=(15, 6)
)
plt.show()
```



Posterior samples

Out-of-sample predictions

Current `post` | Out-of-sample `post`

```
post
```

arviz.InferenceData

- posterior
- sample_stats
- observed_data
- constant_data

Posterior predictive summary

```
az.summary(
  post.posterior_predictive
)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
obs[0]	1.000	0.000	1.000	1.000	0.000	NaN	4000.0	4000.0	NaN
obs[1]	0.454	0.498	0.000	1.000	0.008	0.001	3987.0	3987.0	1.0
obs[2]	0.999	0.027	1.000	1.000	0.000	0.008	4021.0	4000.0	1.0
obs[3]	0.485	0.500	0.000	1.000	0.008	0.000	3785.0	3785.0	1.0
obs[4]	0.008	0.088	0.000	0.000	0.001	0.008	4078.0	4078.0	1.0
...	...	...	...	...	...	...	...	...	...
p[70]	0.577	0.109	0.374	0.782	0.002	0.001	3240.0	2810.0	1.0
p[71]	0.995	0.008	0.983	1.000	0.000	0.000	2655.0	2907.0	1.0
p[72]	0.038	0.027	0.003	0.087	0.001	0.001	953.0	1377.0	1.0
p[73]	0.965	0.035	0.903	1.000	0.001	0.001	2377.0	2163.0	1.0
p[74]	0.856	0.057	0.741	0.947	0.001	0.001	2679.0	2752.0	1.0

150 rows × 9 columns

Evaluation

```
post.posterior["p"].shape
```

```
(4, 1000, 175)
```

```
post.posterior_predictive["p"].shape
```

```
(4, 1000, 75)
```

```
p_train = post.posterior["p"].mean(dim=["chain", "draw"])
p_test  = post.posterior_predictive["p"].mean(dim=["chain", "draw"])
```

```python
print(p_train)
```

```
<xarray.DataArray 'p' (p_dim_0: 175)> Size: 1kB
array([0.93819, 0.26132, 0.01864, 0.99999, 0.465
       0.28936, 0.95823, 0.53909, 0.12185, 0.631
       0.48929, 0.44769, 1.     , 0.25201, 0.418
       0.     , 0.00025, 0.99912, 0.21269, 0.855
       0.00001, 0.58379, 0.90272, 0.91664, 0.001
       0.00006, 0.86009, 1.     , 0.99999, 0.673
       0.64579, 0.11999, 0.44915, 0.17473, 0.813
       0.64672, 0.06343, 0.14008, 1.     , 1.
       0.00011, 0.48045, 0.04294, 0.00011, 1.
       0.43427, 0.99984, 0.00117, 0.99389, 0.809
       0.21644, 0.302  , 0.64459, 0.84646, 0.000
       0.     , 0.00174, 0.29865, 0.48853, 0.843
       0.0127 , 0.99999, 1.     , 0.00036, 0.999
       0.99735, 0.00578, 0.0001 , 1.     , 0.053
       0.0212 , 0.00001, 0.00001, 0.82958, 0.998
       0.00946, 0.     , 0.0855 , 0.94712, 0.410
       0.99999, 0.12598, 0.00345, 0.61756, 0.842
       0.00063, 0.00373, 0.45406, 0.56222, 0.991
       0.55747, 0.18935, 0.90859, 0.40357, 0.866
       0.00303, 0.11039, 0.05068, 0.38197, 0.360
       0.92466, 0.01816, 0.00052, 0.13198, 0.
```

```python
print(p_test)
```

```
<xarray.DataArray 'p' (p_dim_0: 75)> Size: 600B
array([1.     , 0.45196, 0.99975, 0.46804, 0.006
       0.99558, 0.08307, 0.17285, 0.78742, 0.999
       1.     , 0.99499, 1.     , 0.     , 1.
       0.99795, 0.08286, 0.     , 0.36999, 0.888
       0.80057, 0.95772, 0.21037, 0.00643, 0.
       0.21745, 0.1475 , 0.00636, 0.00013, 0.054
       0.55927, 0.     , 0.19585, 0.99921, 0.004
       0.     , 0.73422, 1.     , 1.     , 0.999
       0.92828, 0.43828, 0.99233, 0.01992, 0.999
       0.03784, 0.96468, 0.8558 ])
Coordinates:
  * p_dim_0  (p_dim_0) int64 600B 0 1 2 3 4 5 6
```

ROC & AUC

```python
from sklearn.metrics import RocCurveDisplay, accuracy_score, auc, roc_curve
```

```python
# Test data
fpr_test, tpr_test, thd_test = roc_curve(y_true=y_test, y_score=p_test)
auc_test = auc(fpr_test, tpr_test); auc_test
```

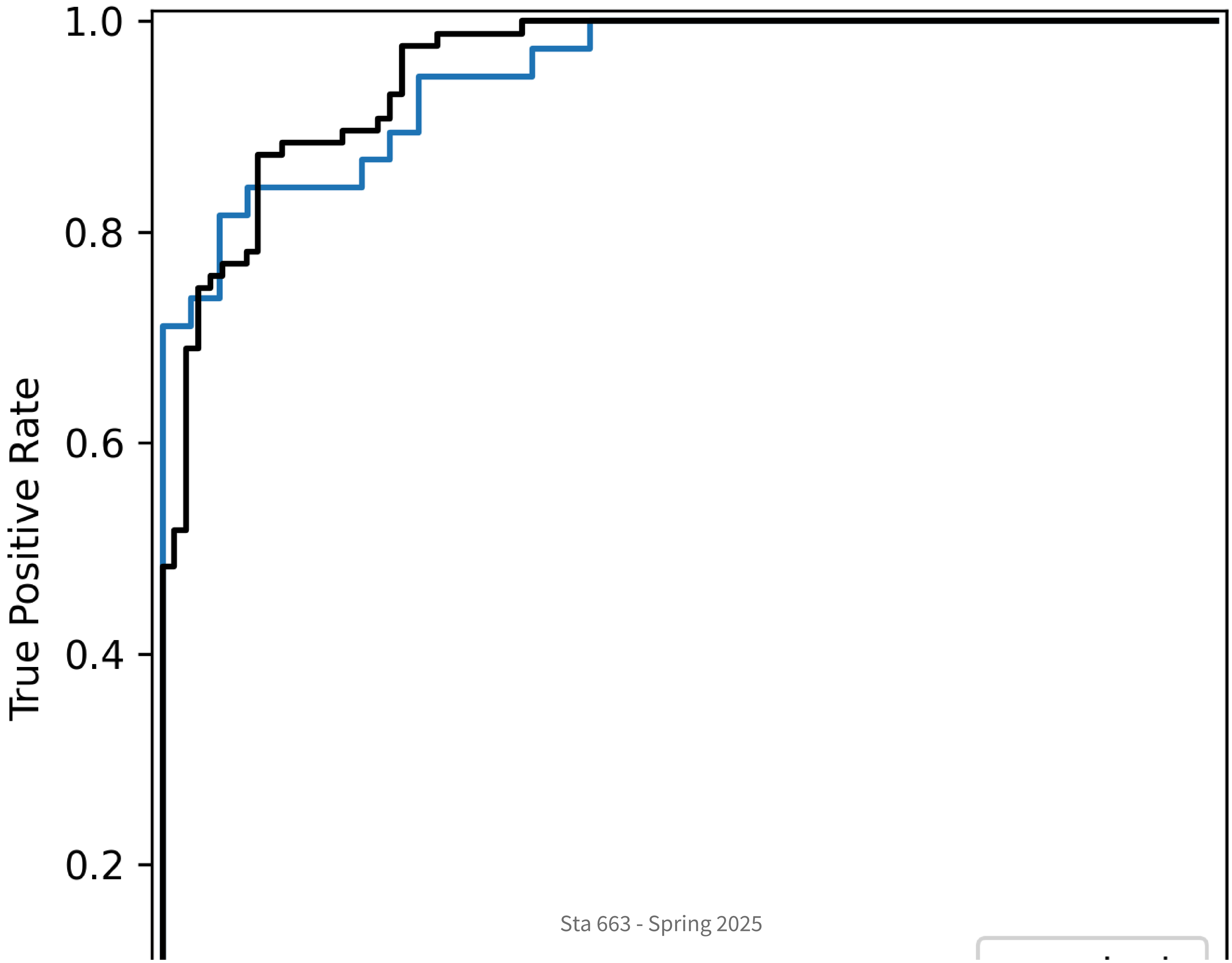
```
np.float64(0.9495021337126599)
```

```python
# Training data
fpr_train, tpr_train, thd_train = roc_curve(y_true=y_train, y_score=p_train)
auc_train = auc(fpr_train, tpr_train); auc_train
```

```
np.float64(0.9553291536050157)
```

ROC Curves

```
fig, ax = plt.subplots()
roc = RocCurveDisplay(fpr=fpr_test, tpr=tpr_test).plot(ax=ax, label="test")
roc = RocCurveDisplay(fpr=fpr_train, tpr=tpr_train).plot(ax=ax, color="k", label="train")
plt.show()
```


1.0
0.8
0.6
0.4
0.2
True Positive Rate