PyMC + ArviZ

Lecture 18

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
print(pm.__version__)
```

```
5.28.1
```

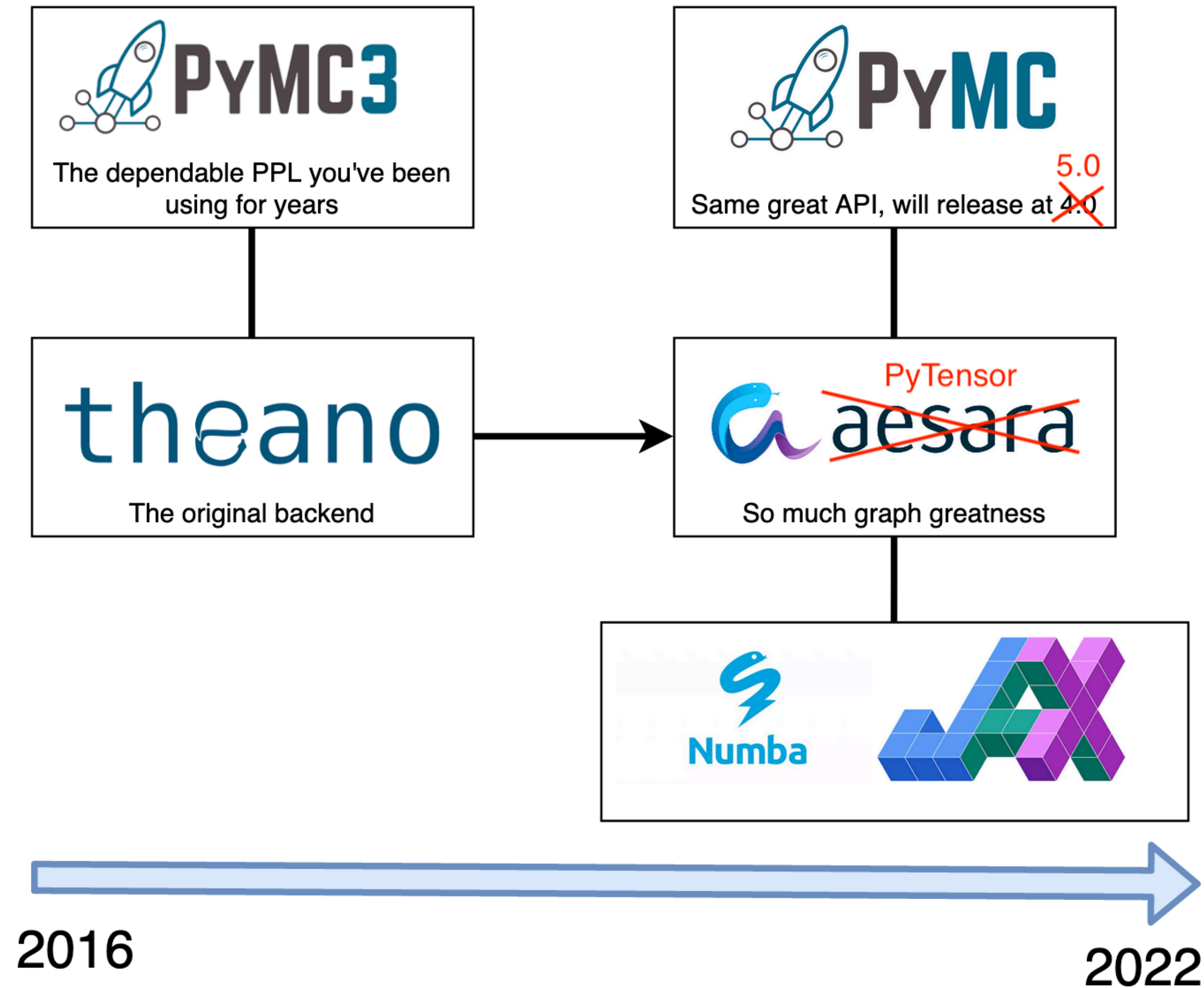
ArviZ

> 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
print(az.__version__)
```

```
0.23.4
```

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 - variables defined inside are added to the parent scope (be careful about overwriting other variables).

```
x
```

x

```
type(x)
```

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

Components are also attached to the model object:

```
norm.x
```

x

```
norm["x"]
```

x

Using a component without a context

PyMC requires a model context to register components - creating a distribution outside a `with` block raises an error:

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

```
TypeError: No model on context stack, which is needed to instantiate distributions. Add variable
inside a 'with model:' block, or use the '.dist' syntax for a standalone distribution.
```

To construct a standalone `TensorVariable` outside a model, use the `.dist()` class method:

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

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

```
array([[ 0.16247,  3.83461, -0.83205],
       [-2.72404,  2.20162,  0.45156]])
```

```
1  pm.logp(z, 0.)
```

```
Alloc.0
```

Random Variables

PyMC distributions are implemented as `TensorVariable` objects - some useful attributes and functions:

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

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

```
1  norm.x.name
```

```
'x'
```

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

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

```
1  norm.x.owner.inputs
```

```
[RNG(<Generator(PCG64) at 0x10F407920>), Constant(<pytensor.tensor.type_other.NoneTypeT object at
```

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

```
array(0.80662)
```

```
1  pm.logp(norm.x, 0.)
```

```
x_logprob
```

Modifying models

This context construction makes it possible to add additional components to an existing (named) model via subsequent `with` statements

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

```
norm.basic_RVs
```

```
[x, y]
```

Variable hierarchy

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

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

```
array([-1.64614, -0.27549, -0.66076])
```

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

```
array([[-1.03779, -0.18752, -0.45454],
       [ 0.31004, -1.77494, -0.10683],
       [-0.00118, -0.75024, -0.20394],
       ...,
       [-0.76928,  0.0199 , -2.42434],
       [-0.88935,  1.93616,  0.59692],
       [-0.57633,  0.55113, -0.63961]], sh
```

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

```
np.float64(0.03792295292232359 5)
```

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

```
np.float64(1.976885032820083)
```

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

```
np.float64(1.4060174368833707)
```

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)
```

Note the use of `observed` - this fixes the variable's value to the provided data (i.e. it is not sampled). This is how we condition models on observed data.

```
beta_binom.basic_RVs
```

```
[p, x]
```

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

```
1 type(trace)
```

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

```
1 trace
```

arviz.InferenceData

- ▸ posterior
- ▸ sample_stats
- ▸ observed_data

Xarray - N-D labeled arrays and datasets in Python

> Xarray makes working with labelled multi-dimensional arrays in Python 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 integrates tightly with dask for parallel computing.

Digging into `trace`

```
1 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:                 2026-03-17T13:49:16.801350+00:00
    arviz_version:              0.23.4
    inference_library:          pymc
    inference_library_version:  5.28.1
    sampling_time:              0.19043397903442383
    tuning_steps:               1000
```

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

```
(4, 1000)
```

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

```
(1000,)
```

```
1 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()
```

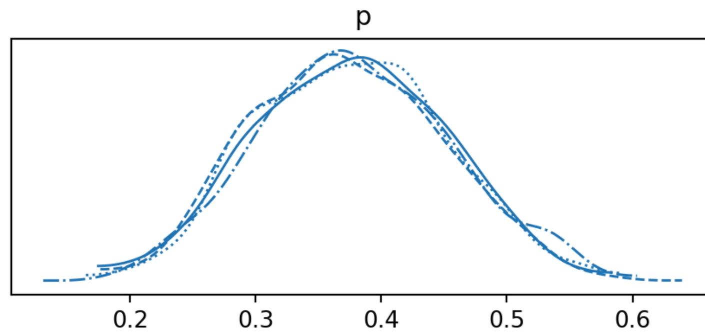
```
                     p
chain draw
0     0     0.334673
      1     0.343498
      2     0.262910
      3     0.346714
      4     0.288526
...              ...
3     995   0.421379
      996   0.432828
      997   0.327570
      998   0.327570
      999   0.348614

[4000 rows x 1 columns]
```

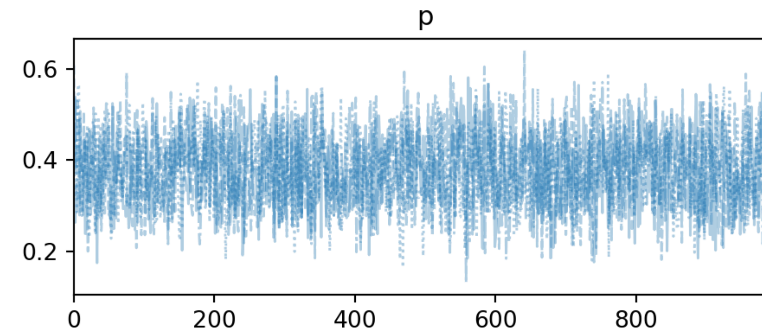
```python
trace.posterior["p"][0,:].to_dataframe()
```

```
      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 x 2 columns]
```

Traceplots with ArviZ

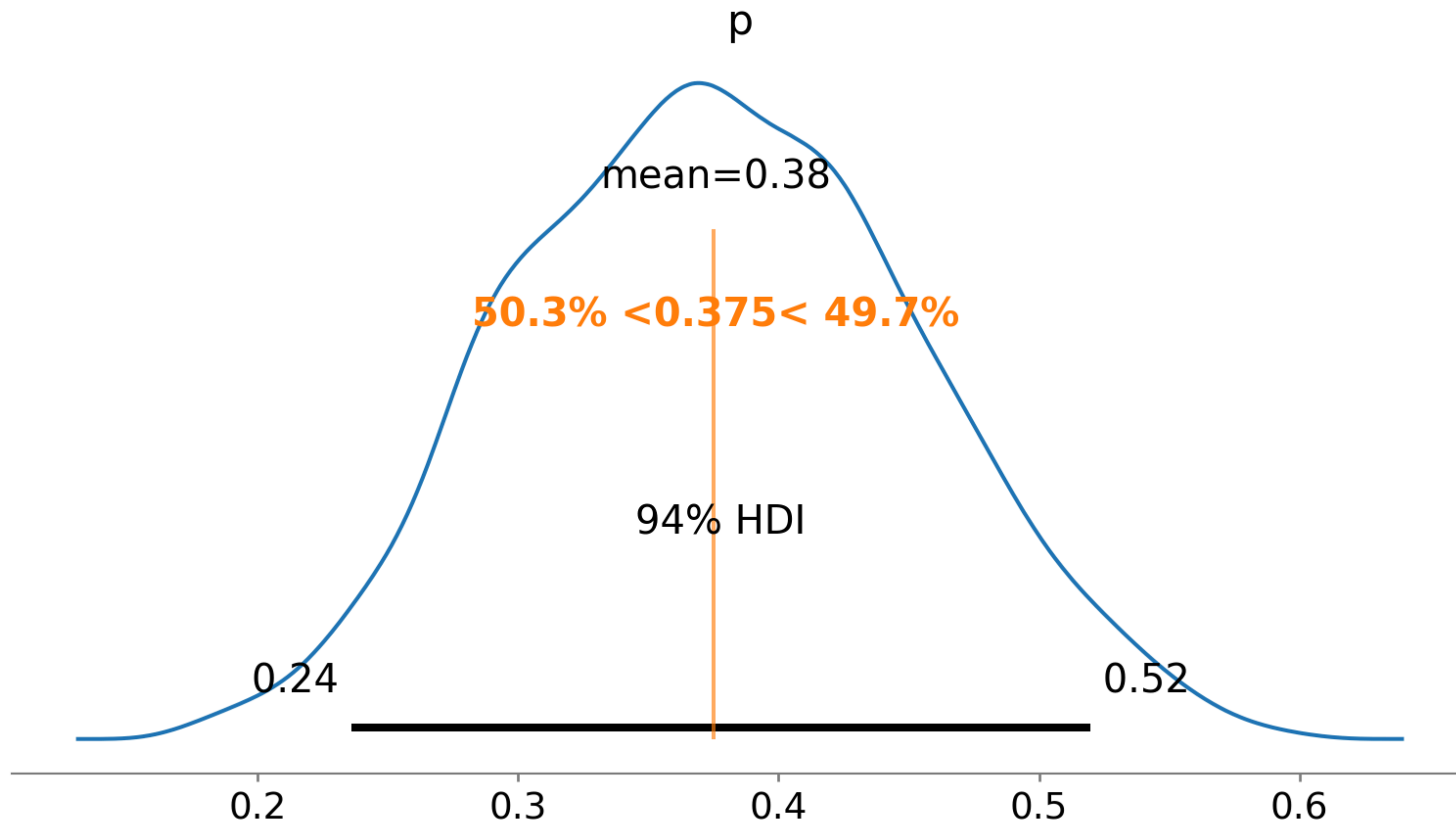
```
1 axs = az.plot_trace(trace)
2 plt.show()
```



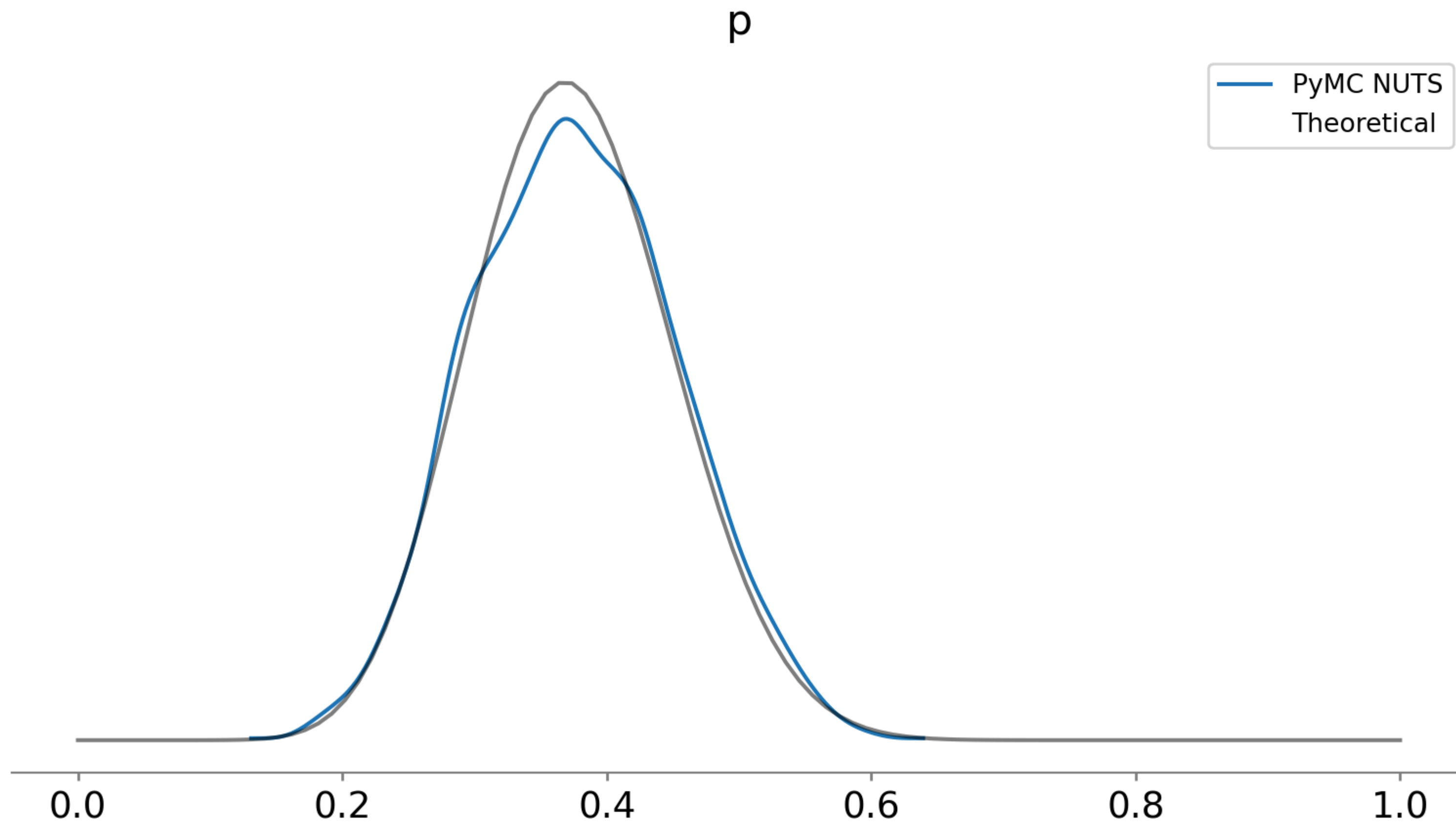


Posterior plot with ArviZ

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

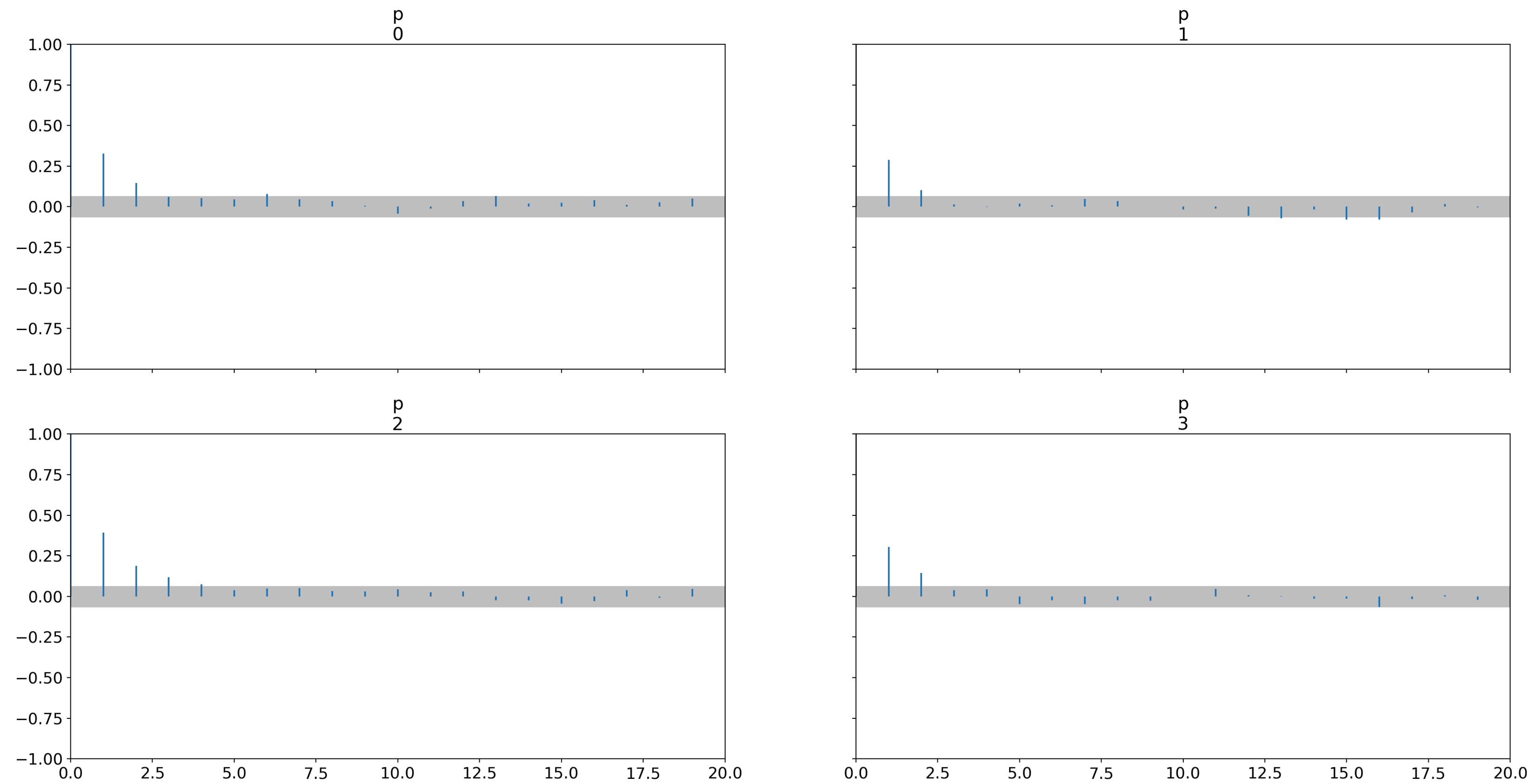


PyMC vs Theoretical



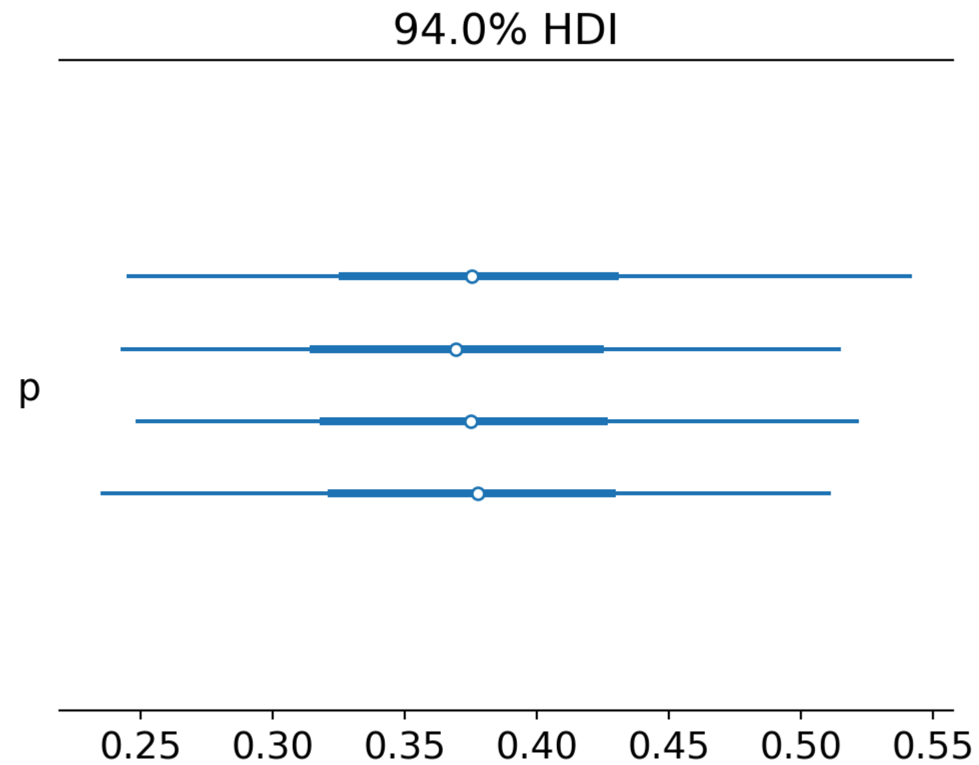
Autocorrelation plots

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



Forest plots

```
axs = 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  0.376  0.077   0.236     0.52      0.002    0.001    1757.0    2546.0    1.0
```

individual methods are available for each statistic,

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

```
<xarray.DataArray 'p' ()> Size: 8B
array(1756.93606)
```

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

```
<xarray.DataArray 'p' ()> Size: 8B
array(1.0005)
```

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

```
<xarray.DataArray 'p' ()> Size: 8B
array(2546.47826)
```

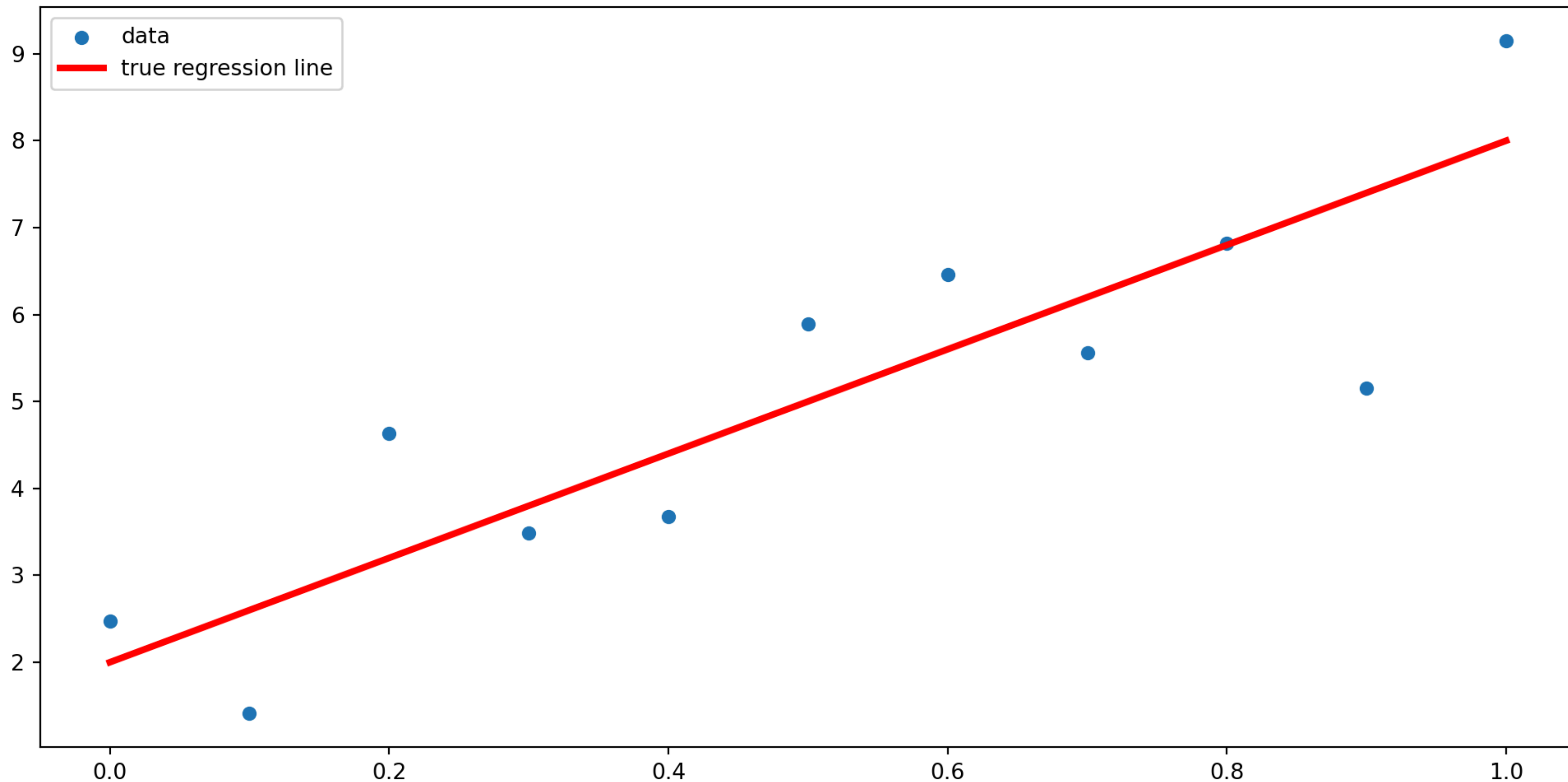
```python
print(az.mcse(trace).p)
```

```
<xarray.DataArray 'p' ()> Size: 8B
array(0.00184)
```

Demo 1 - Linear regression

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

```
1 with pm.Model() as lm:
2   m = pm.Normal('m', mu=0, sigma=50)
3   b = pm.Normal('b', mu=0, sigma=50)
4   sigma = pm.HalfNormal('sigma', sigma=5)
5 
6   likelihood = pm.Normal('y', mu=m*x + b, sigma=sigma, observed=y)
7 
8   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

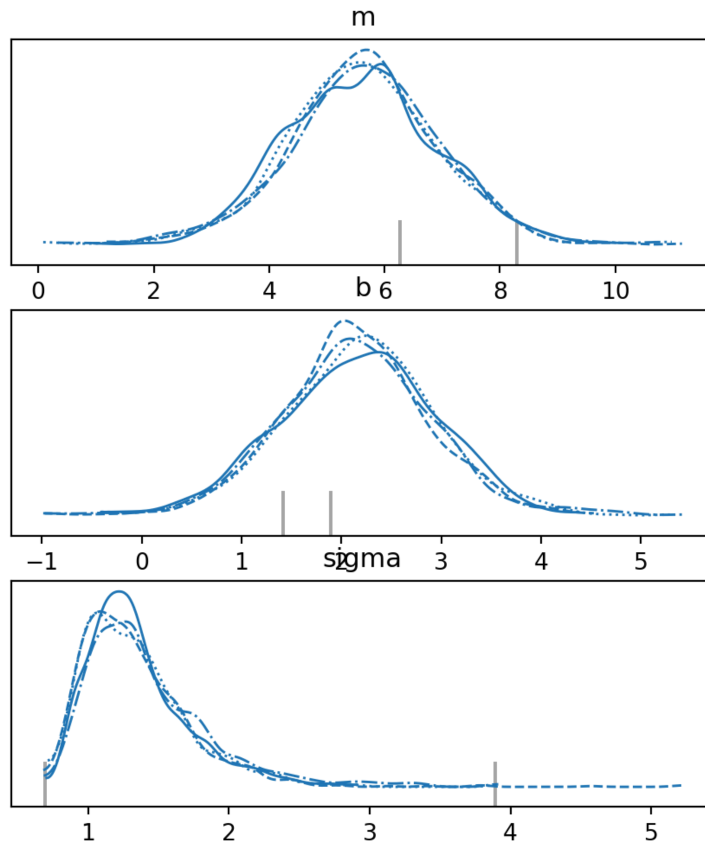
Posterior summary

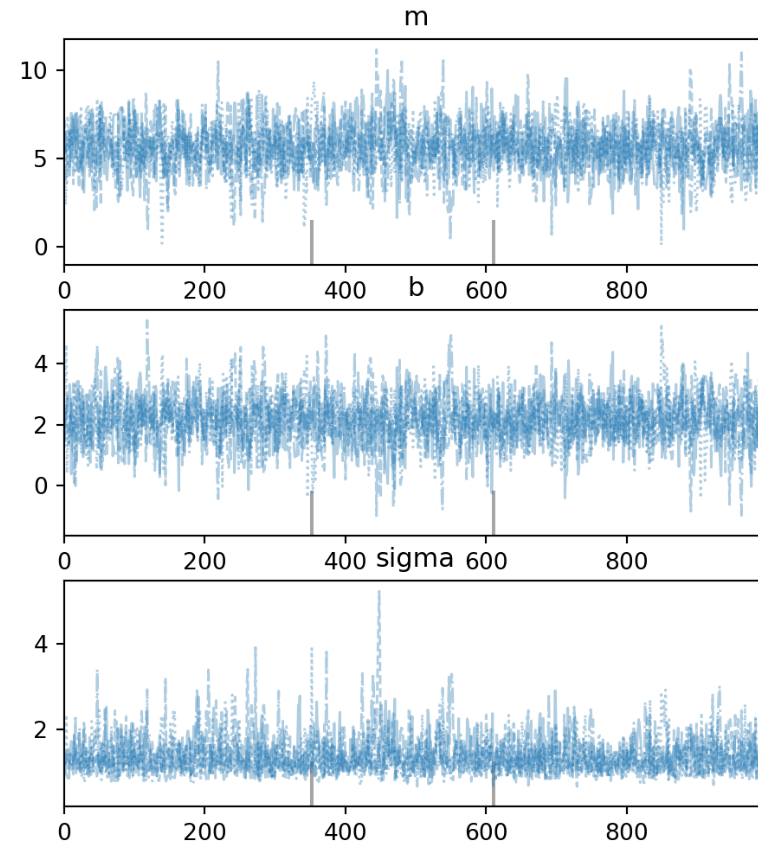
```
1 az.summary(trace)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
m	5.629	1.329	3.145	8.093	0.035	0.029	1418.0	1755.0	1.00
b	2.156	0.775	0.670	3.572	0.020	0.018	1458.0	1568.0	1.00
sigma	1.364	0.405	0.799	2.116	0.013	0.017	1192.0	1541.0	1.01

Trace plots

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

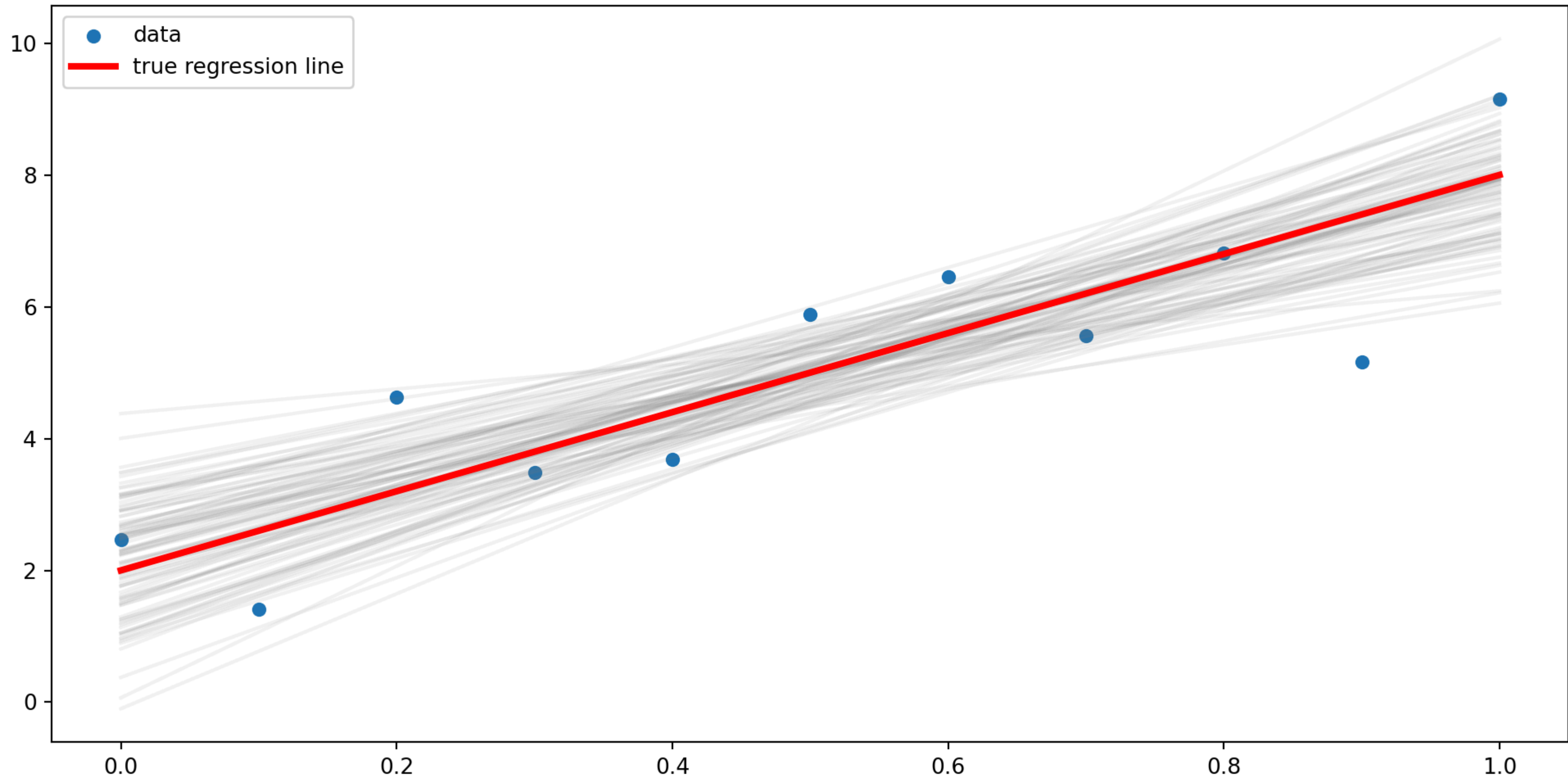




Regression line posterior draws

```
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)
```

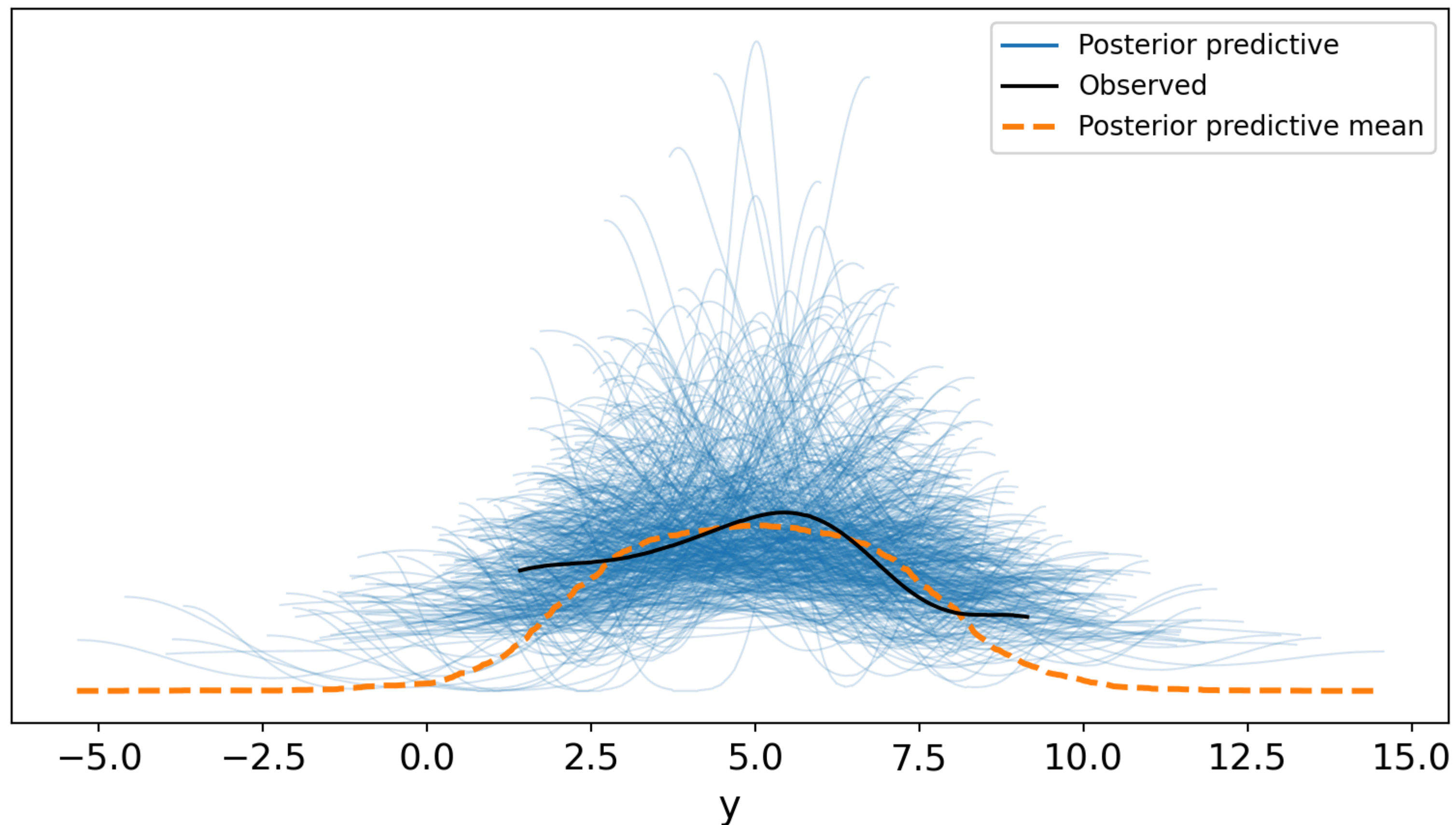
```
pp
```

arviz.InferenceData

- ▸ posterior_predictive
- ▸ observed_data

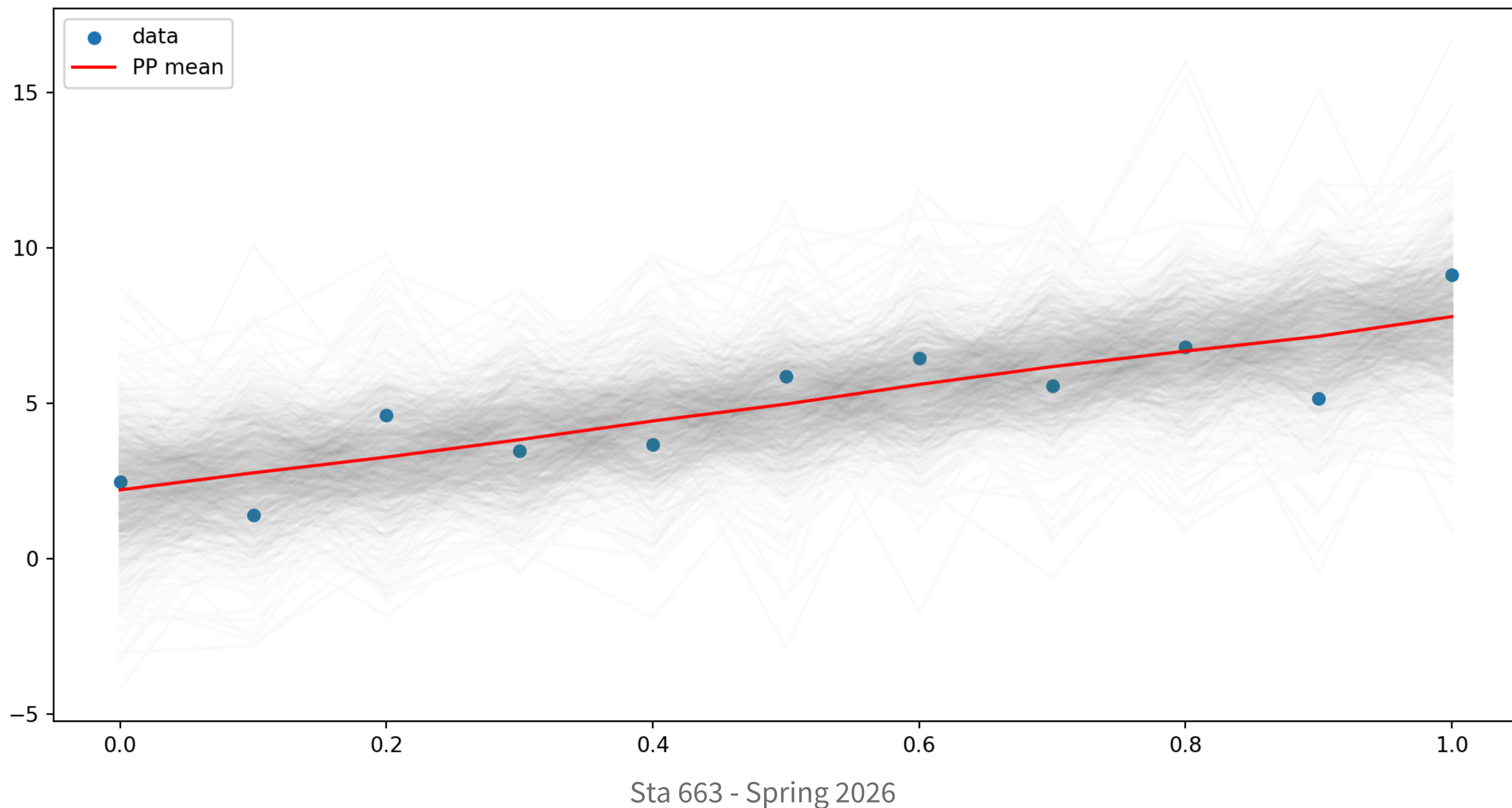
Plotting the posterior predictive distribution

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



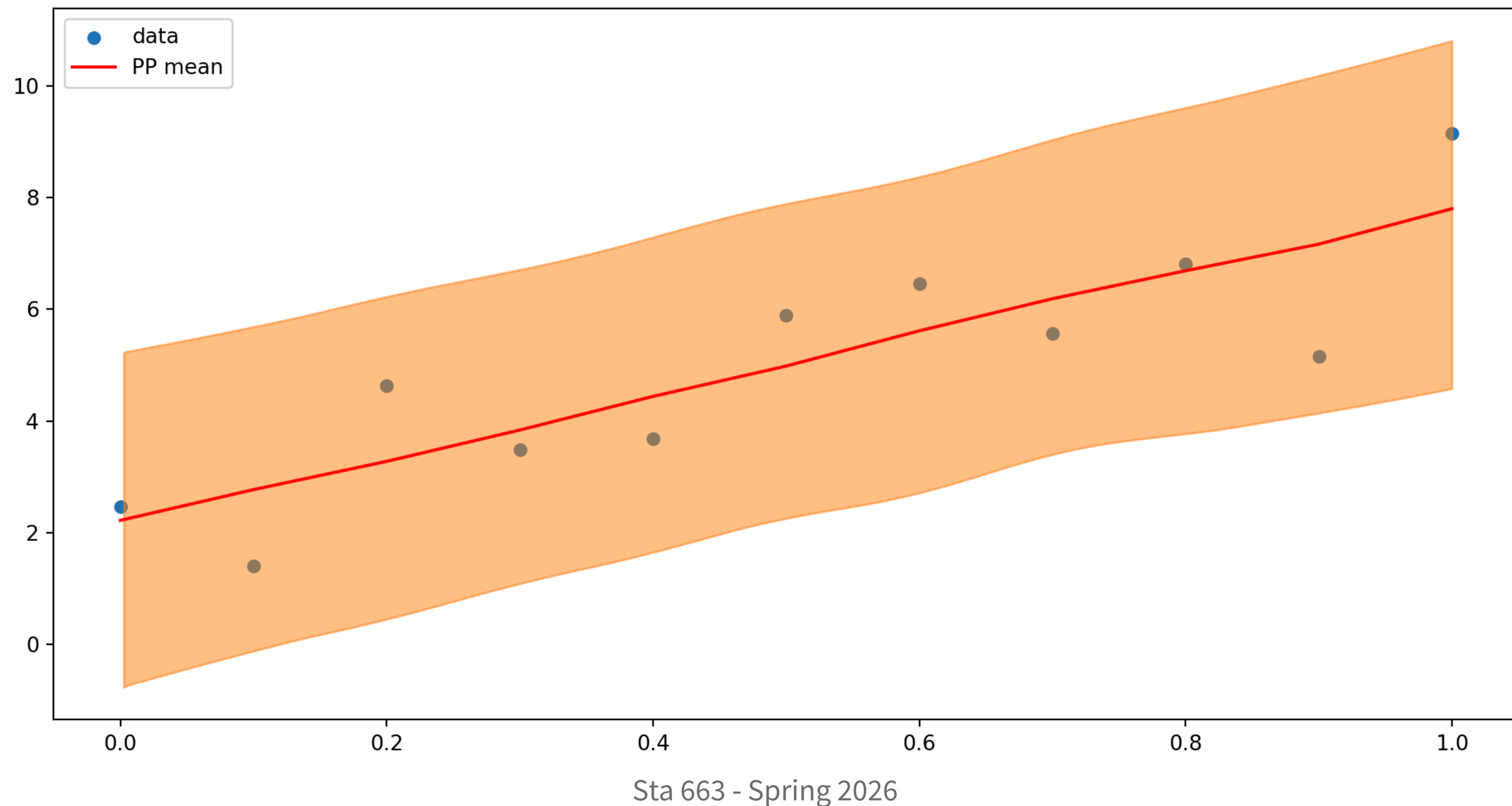
PP draws

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



PP HDI

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



Model revision

By wrapping `m*x + b` in `pm.Deterministic()` we can track this derived quantity in the trace and generate posterior predictive draws for the mean function (rather than just the observation-level predictions).

```
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=False
  )
```

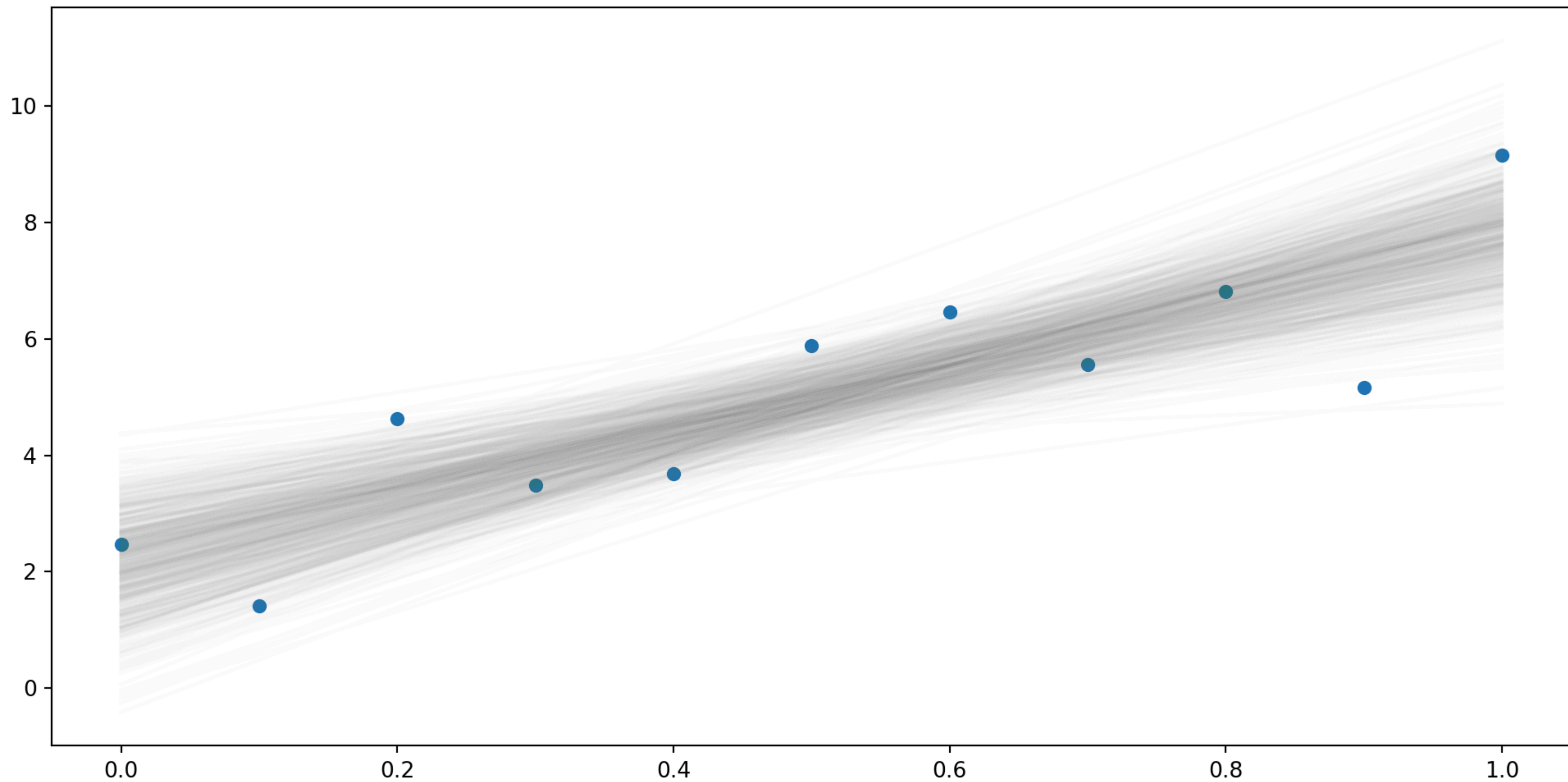
$\hat{y}$ - PP

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_ha
m	5.629	1.329	3.145	8.093	0.035	0.029	1418.0	1755.0	1.0
b	2.156	0.775	0.670	3.572	0.020	0.018	1458.0	1568.0	1.0
sigma	1.364	0.405	0.799	2.116	0.013	0.017	1192.0	1541.0	1.0
y_hat[0]	2.156	0.775	0.670	3.572	0.020	0.018	1458.0	1568.0	1.0
y_hat[1]	2.719	0.667	1.499	3.982	0.017	0.015	1544.0	1633.0	1.0
y_hat[2]	3.282	0.568	2.180	4.291	0.014	0.012	1712.0	1919.0	1.0
y_hat[3]	3.845	0.487	2.914	4.743	0.011	0.010	2067.0	2144.0	1.0
y_hat[4]	4.408	0.432	3.566	5.204	0.008	0.008	2869.0	2429.0	1.0
y_hat[5]	4.971	0.414	4.155	5.741	0.006	0.008	4130.0	2799.0	1.0
y_hat[6]	5.534	0.438	4.719	6.382	0.007	0.009	4394.0	2752.0	1.0
y_hat[7]	6.097	0.497	5.170	7.061	0.008	0.011	3573.0	2550.0	1.0
y_hat[8]	6.660	0.581	5.615	7.809	0.011	0.012	2821.0	2475.0	1.0
y_hat[9]	7.222	0.681	6.026	8.583	0.014	0.014	2372.0	2506.0	1.0
y_hat[10]	7.785	0.791	6.289	9.284	0.017	0.016	2113.0	2517.0	1.0

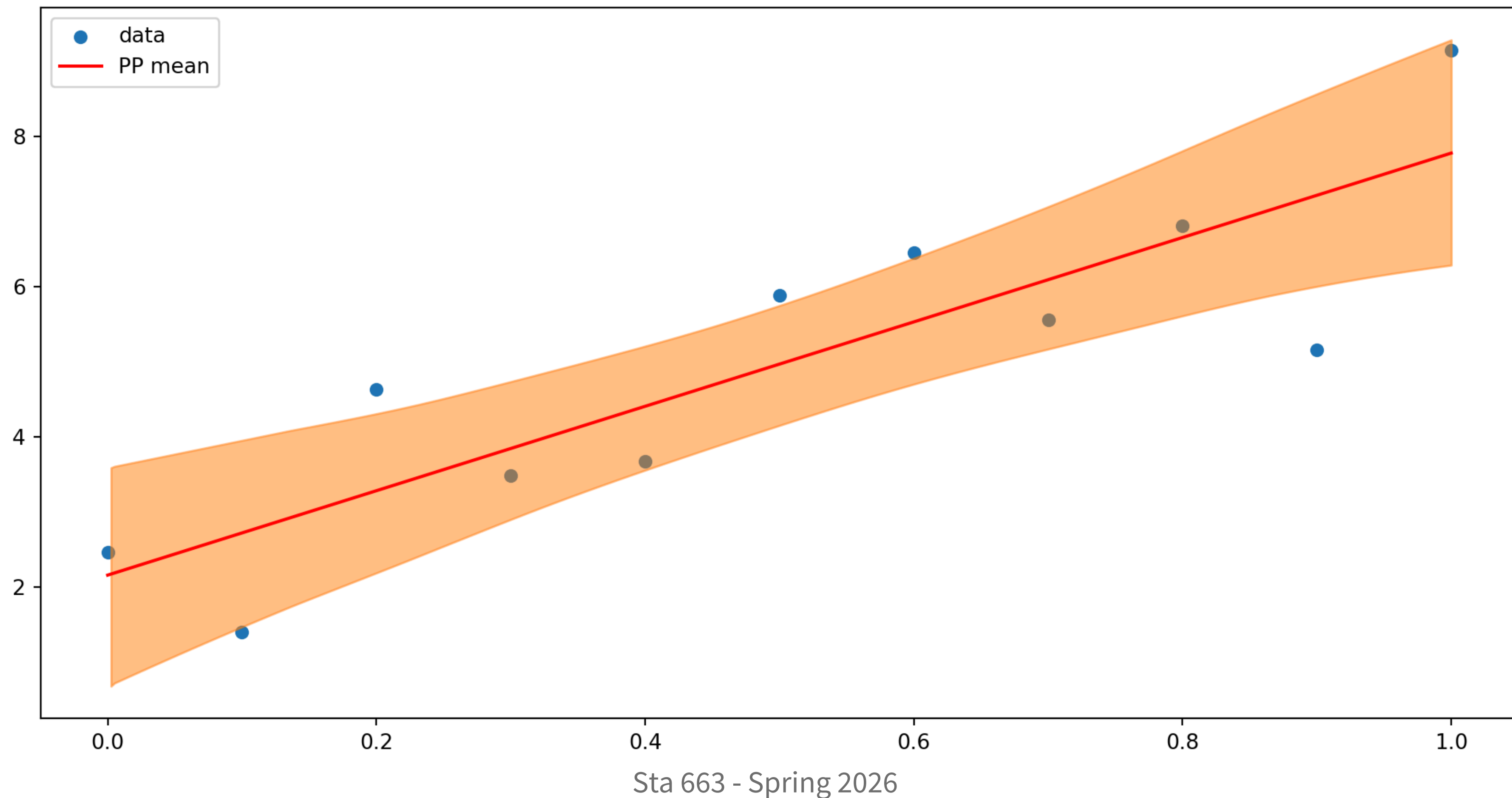
$\hat{y}$ - PP draws

```
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()
```



Demo 2 - Bayesian Lasso

```python
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)

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

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

Note any warnings from the sampler - divergences indicate the sampler had difficulty exploring the posterior (often due to poorly specified priors or model geometry). Large numbers of divergences suggest the results should not be trusted.

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

```
              mean        sd     hdi_3%    hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_ta
beta[0]   -230.110  1051.471  -2122.764   1588.034    506.354  264.540       5.0      17
beta[1]    189.346  1534.946  -1467.009   3398.762    731.692  624.755       5.0      11
beta[2]   -297.668   457.031  -1001.502    479.395    215.023   73.083       5.0      11
beta[3]   -358.025  1054.157  -1871.283   1643.062    511.156  275.699       5.0      11
beta[4]   -460.791   658.009  -1687.435    373.611    303.674  170.469       5.0      11
...            ...       ...        ...        ...        ...      ...       ...       .
beta[96] -1016.168   686.708  -2248.531   -145.503    330.527  140.027       5.0      13
beta[97]  -221.856   482.609  -1240.211    619.665    208.244  117.138       5.0      14
beta[98]  -192.017   716.281  -1649.817    779.600    337.012  188.262       5.0      11
beta[99]  -671.608   833.620  -2339.851    328.992    381.139  145.968       5.0      25
sigma        2.115     1.304      0.202      4.396      0.510    0.142       7.0      17

[101 rows x 9 columns]
```

Weakly informative model

```
1 with pm.Model() as bayes_weak:
2   b = pm.Normal("beta", mu=0, sigma=10, shape=k)
3   s = pm.HalfNormal('sigma', sigma=2)
4 
5   likelihood = pm.Normal("y", mu=X @ b, sigma=s, observed=y)
6 
7   trace = pm.sample(progressbar=False, random_seed=12345)
```

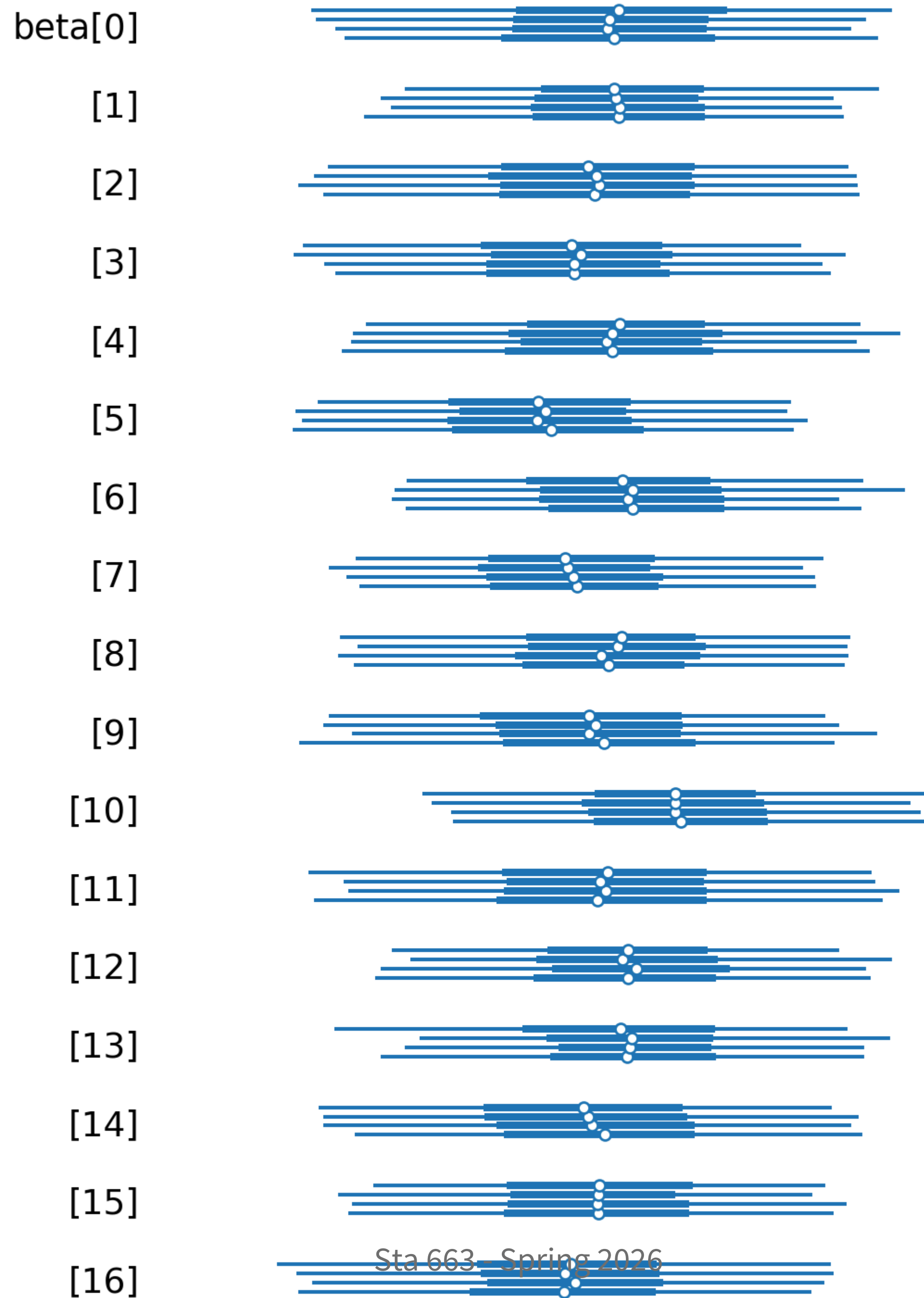
```
1 az.summary(trace)
```

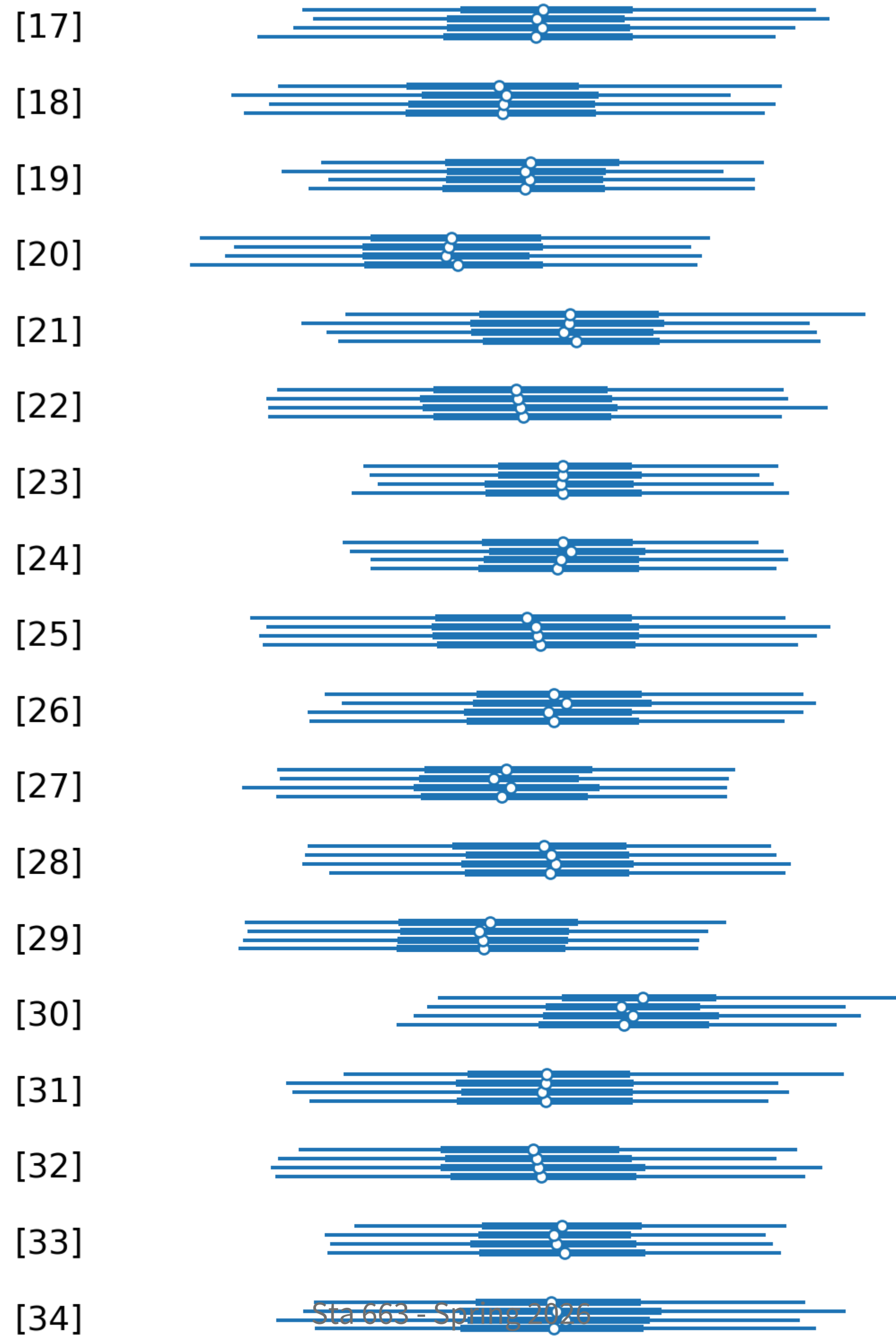
```
            mean     sd   hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
beta[0]    0.585  7.763 -13.738   15.280      0.103    0.125    5836.0    3052.0   1.00
beta[1]    1.061  6.676 -11.240   13.635      0.094    0.120    5007.0    2739.0   1.00
beta[2]   -0.271  7.631 -14.916   13.594      0.101    0.151    5684.0    2330.0   1.00
beta[3]   -1.331  7.216 -15.468   11.740      0.112    0.135    4229.0    1168.0   1.00
beta[4]    0.772  7.525 -13.477   14.330      0.098    0.123    5841.0    3134.0   1.00
...          ...    ...      ...      ...        ...      ...       ...       ...    ...
beta[96]  -0.111  6.579 -12.839   12.116      0.100    0.106    4317.0    2591.0   1.00
beta[97]  -0.874  6.673 -13.067   11.592      0.094    0.105    4989.0    2864.0   1.00
beta[98]   1.392  6.728 -10.798   14.804      0.109    0.117    3787.0    2665.0   1.00
beta[99]  -1.180  6.845 -14.337   11.251      0.093    0.119    5449.0    2653.0   1.00
sigma      2.145  1.083   0.606    4.136      0.128    0.058      64.0      98.0   1.07

[101 rows x 9 columns]
```

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

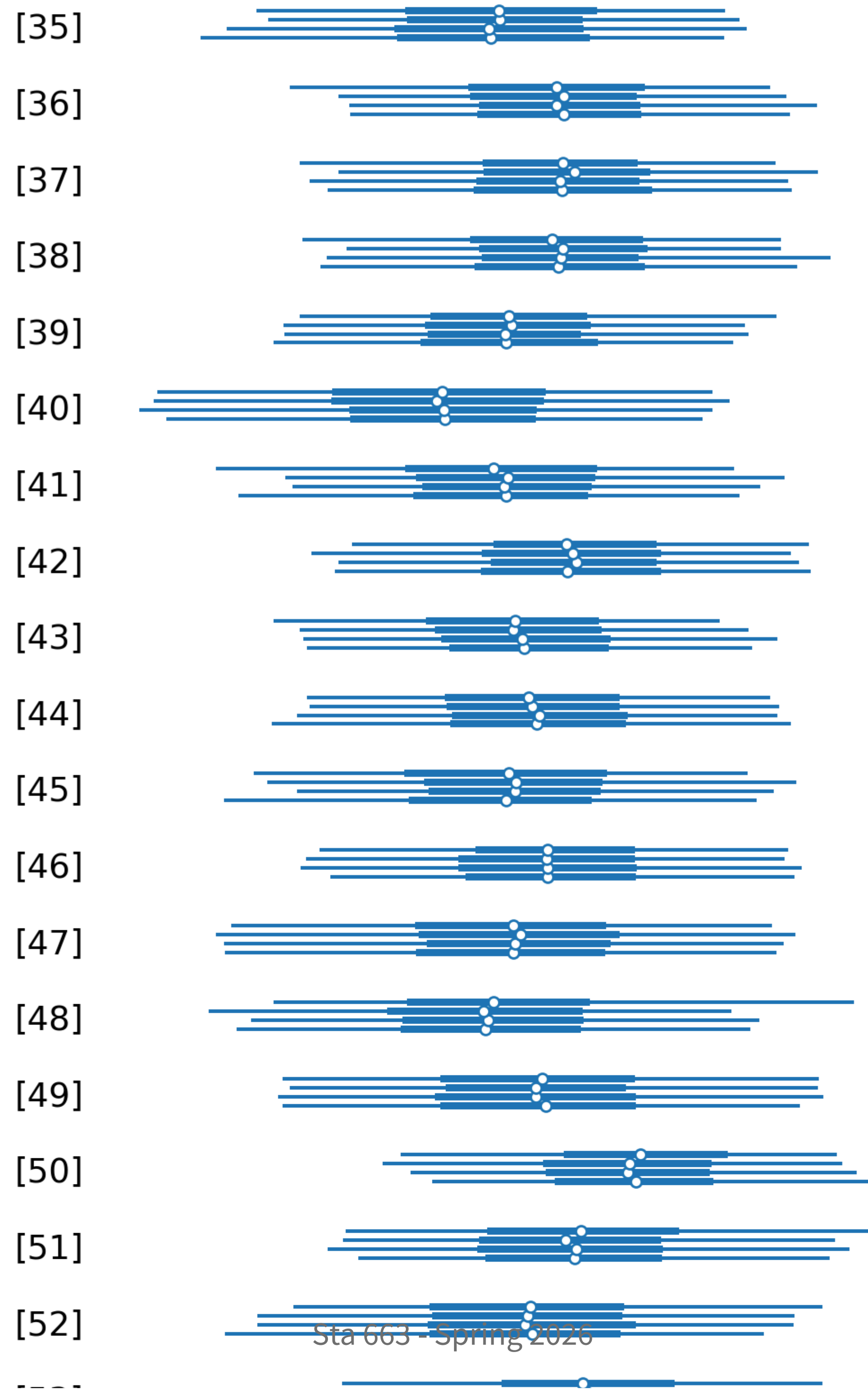
```
            mean     sd   hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
beta[10]   4.042  6.840   -9.114   16.509      0.102    0.127    4538.0    2360.0    1.0
beta[20]  -4.228  7.184  -17.563    9.451      0.110    0.131    4258.0    1975.0    1.0
beta[30]   5.523  6.625   -6.166   18.190      0.128    0.134    2738.0    1067.0    1.0
beta[40]  -4.958  8.103  -20.187   10.227      0.111    0.159    5370.0    2778.0    1.0
beta[50]   5.478  6.546   -6.357   17.987      0.094    0.102    4855.0    2791.0    1.0
beta[60]  -5.773  6.937  -18.517    7.836      0.105    0.115    4403.0    2725.0    1.0
beta[70]   4.751  7.121   -8.624   18.226      0.094    0.120    5739.0    2989.0    1.0
beta[80]  -7.812  6.105  -19.841    2.702      0.098    0.102    3869.0    2779.0    1.0
```

```
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]
[51]
[52]

[53]

[54]

[55]

[56]

[57]

[58]

[59]

[60]

[61]

[62]

[63]

[64]

[65]

[66]

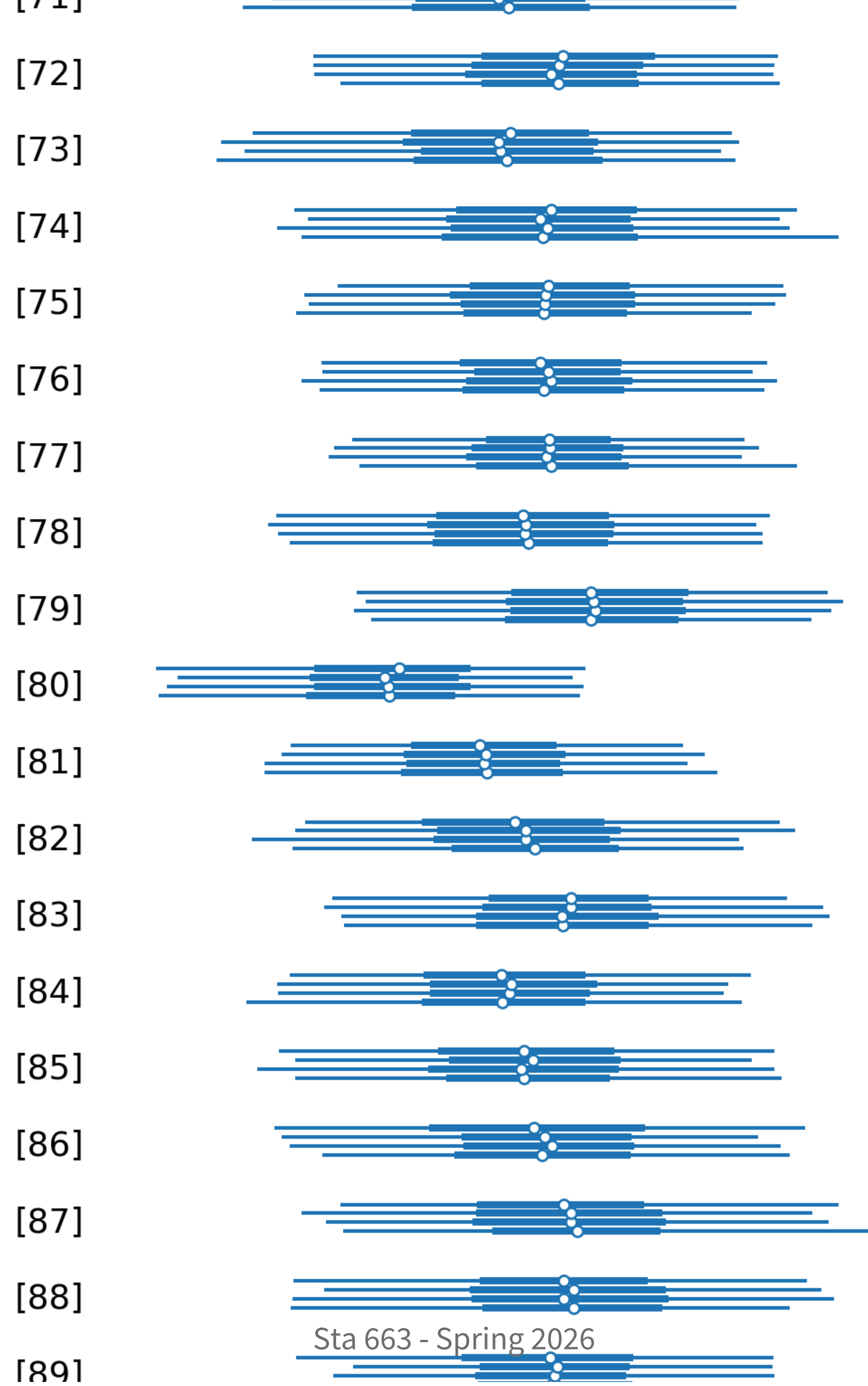
[67]

[68]

[69]

[70]

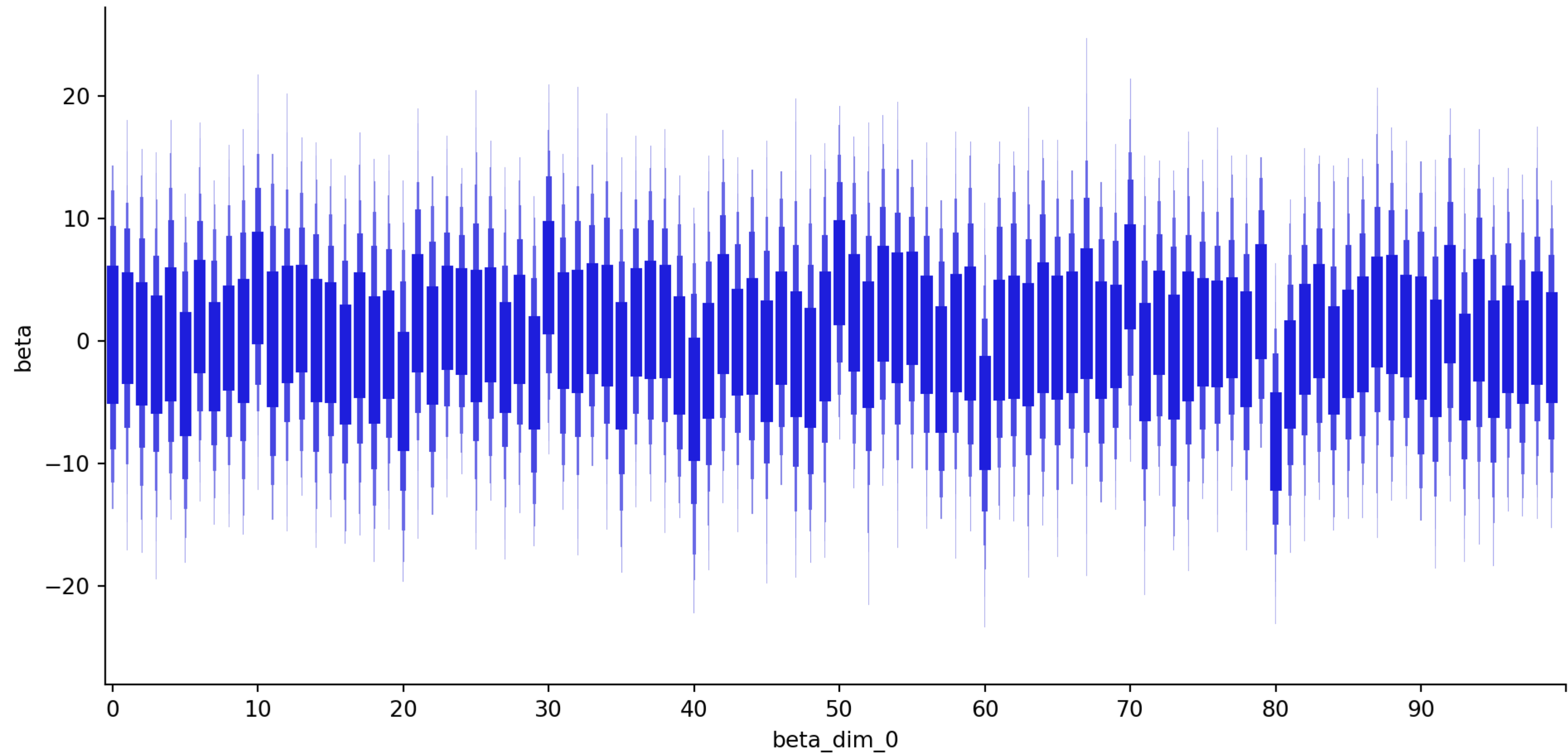
[71]


[72]
[73]
[74]
[75]
[76]
[77]
[78]
[79]
[80]
[81]
[82]
[83]
[84]
[85]
[86]
[87]
[88]
[89]

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, showfliers=False
  )
  plt.tight_layout()
  plt.xticks(range(0,110,10))
  plt.show()

```

```
1 plot_slope(trace)
```



Laplace Prior

Using a Laplace distribution as our prior is the Bayesian analogue of L1 (Lasso) regularization — the resulting MAP estimate is equivalent to the frequentist Lasso solution.

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

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

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

We are ignoring the issue of hyperparameter selection for the Laplace prior here - it enters via the choice of the scale

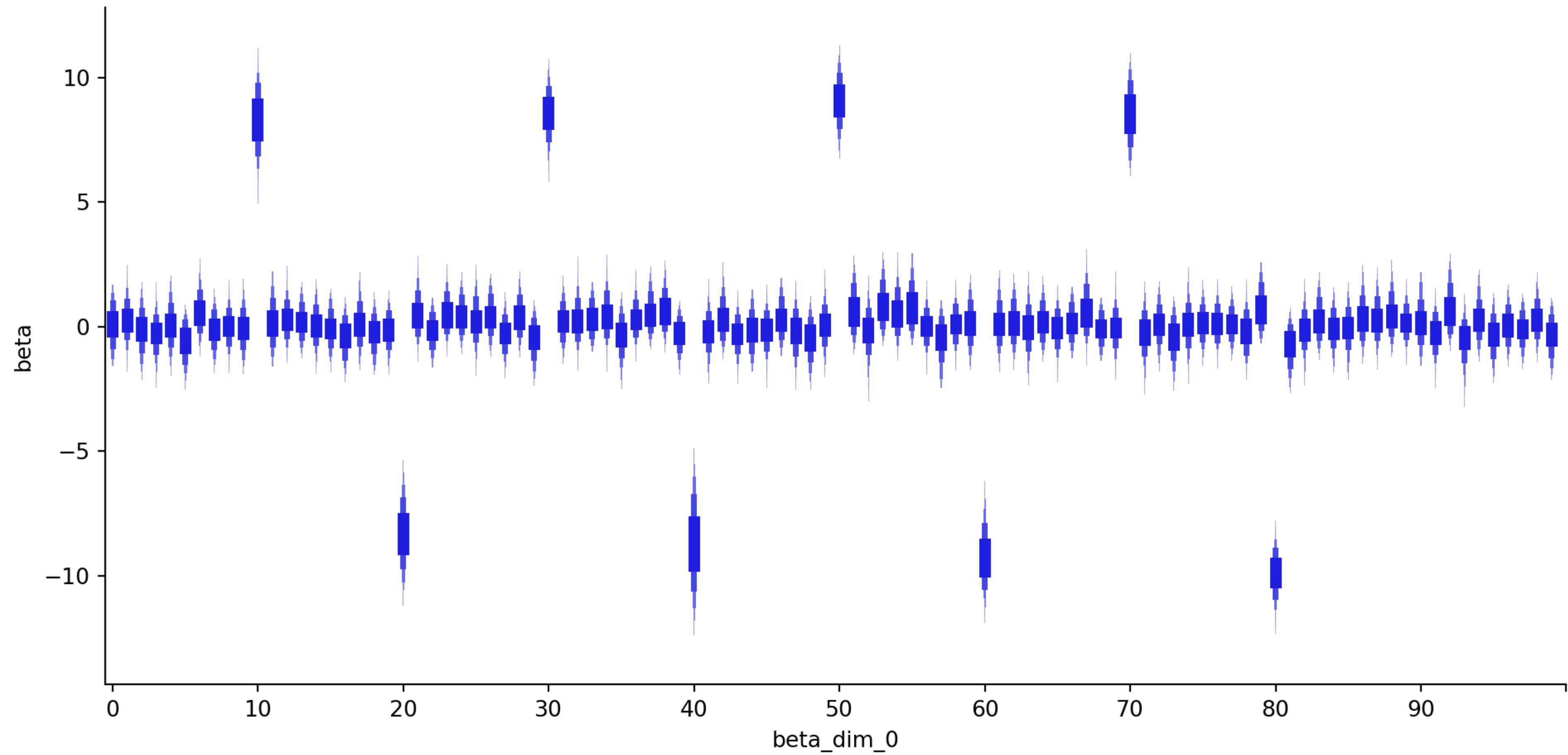
```
az.summary(trace)
```

```
           mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
beta[0]   0.133  0.768  -1.636    1.532      0.019    0.025     757.0    1910.0   1.02
beta[1]   0.289  0.715  -1.072    1.757      0.032    0.019     430.0    1677.0   1.01
beta[2]  -0.105  0.774  -1.857    1.275      0.013    0.026    3599.0    1912.0   1.14
beta[3]  -0.355  0.764  -1.776    1.141      0.071    0.020      93.0    1647.0   1.03
beta[4]   0.098  0.776  -1.422    1.636      0.018    0.041    1729.0    1493.0   1.14
...         ...    ...     ...      ...        ...      ...       ...       ...    ...
beta[96]  0.031  0.670  -1.232    1.421      0.014    0.039    2116.0    1987.0   1.15
beta[97] -0.088  0.663  -1.468    1.120      0.020    0.027    1090.0     427.0   1.02
beta[98]  0.422  0.782  -1.020    1.827      0.090    0.020      83.0     480.0   1.04
beta[99] -0.399  0.721  -1.686    1.015      0.031    0.025     492.0    2385.0   1.02
sigma     0.746  0.450   0.295    1.585      0.080    0.048      17.0       8.0   1.17

[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]   7.986  1.447   5.751   10.352      0.317    0.144      27.0       9.0   1.09
beta[20]  -7.990  1.544 -10.314   -5.499      0.369    0.177      23.0     578.0   1.12
beta[30]   8.706  0.910   6.931   10.416      0.033    0.033     948.0    2160.0   1.04
beta[40]  -9.160  1.743 -11.683   -5.894      0.307    0.031      41.0     242.0   1.07
beta[50]   8.727  1.231   6.742   10.640      0.282    0.151      26.0       9.0   1.10
beta[60]  -9.199  1.079 -11.193   -7.066      0.039    0.038     795.0    2021.0   1.03
beta[70]   8.503  1.094   6.439   10.509      0.041    0.046     896.0    2280.0   1.03
beta[80]  -9.800  0.854 -11.451   -8.204      0.061    0.024     237.0    1933.0   1.02
```

```
1 plot_slope(trace)
```

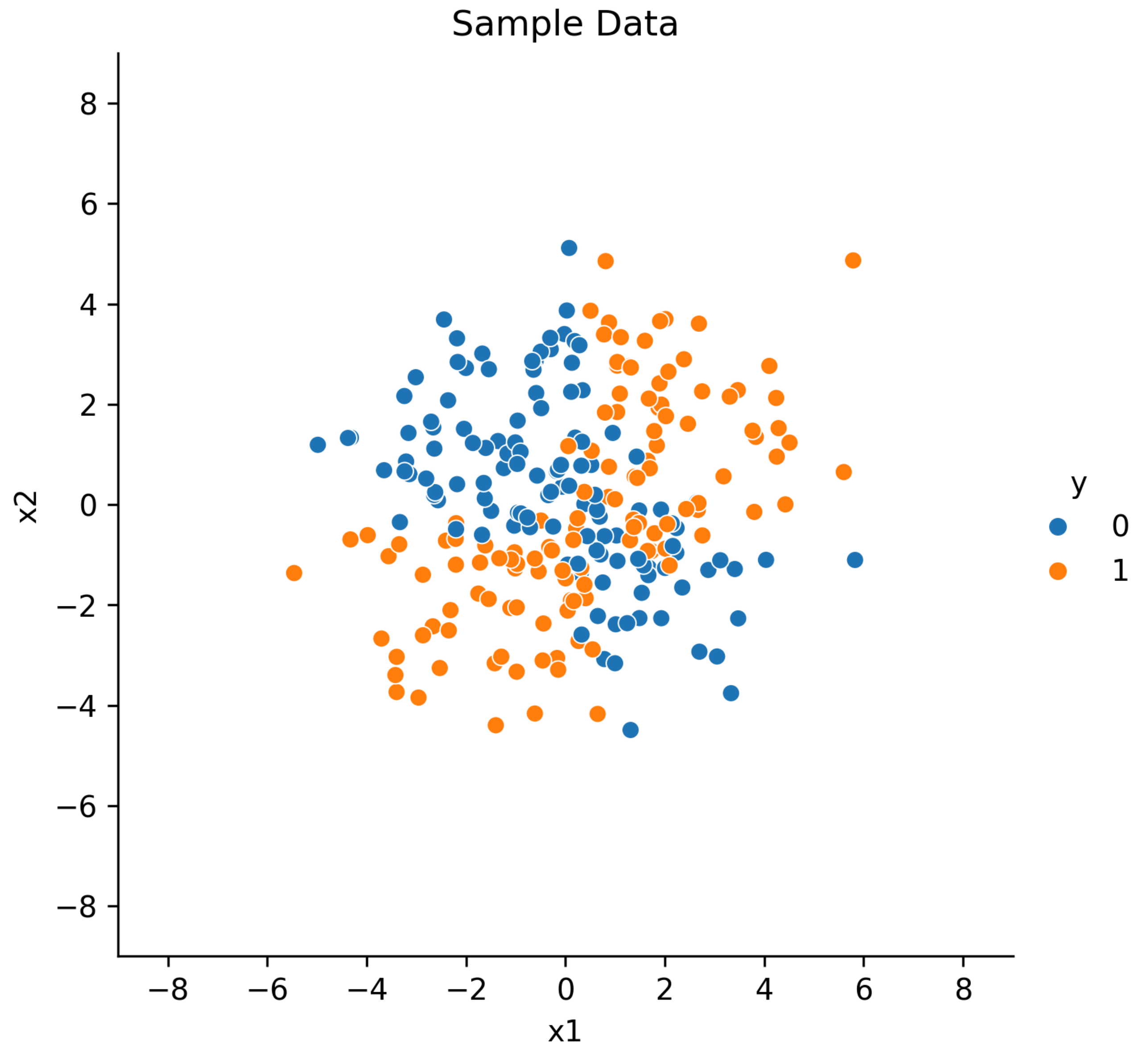


Demo 3 - Logistic Regression

Based on PyMC Out-Of-Sample Predictions example

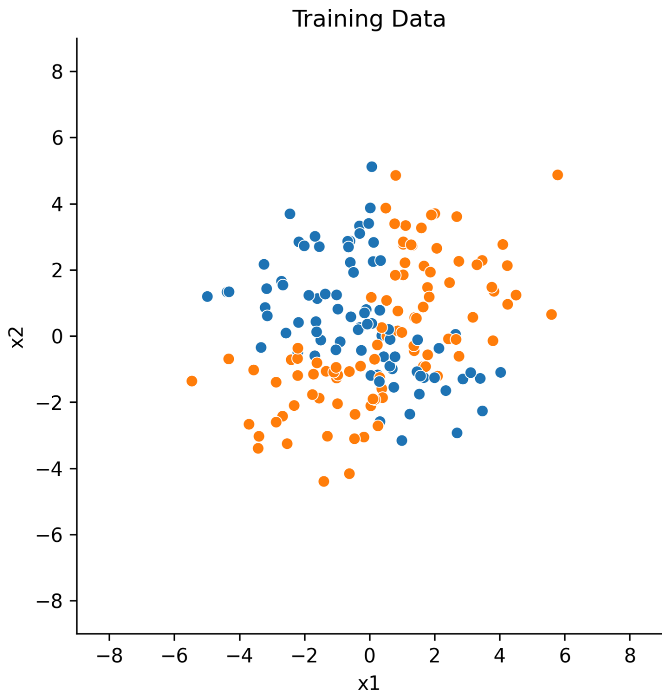
Data

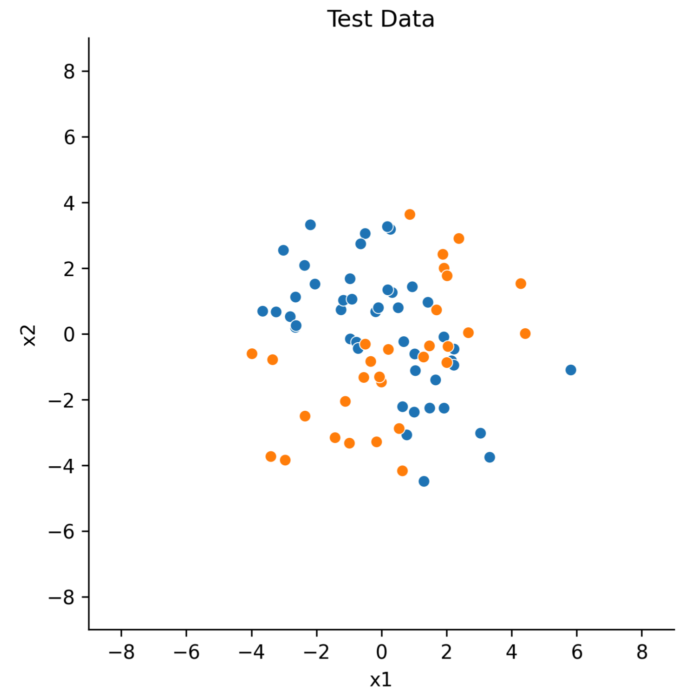
```
           x1        x2  y
0   -3.207674  0.859021  0
1    0.128200  2.827588  0
2    1.481783 -0.116956  0
3    0.305238 -1.378604  0
4    1.727488 -0.926357  1
..        ...       ... ..
245 -2.182813  3.314672  0
246 -2.362568  2.078652  0
247  0.114571  2.249021  0
248  2.093975 -1.212528  1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

[250 rows x 3 columns]
```

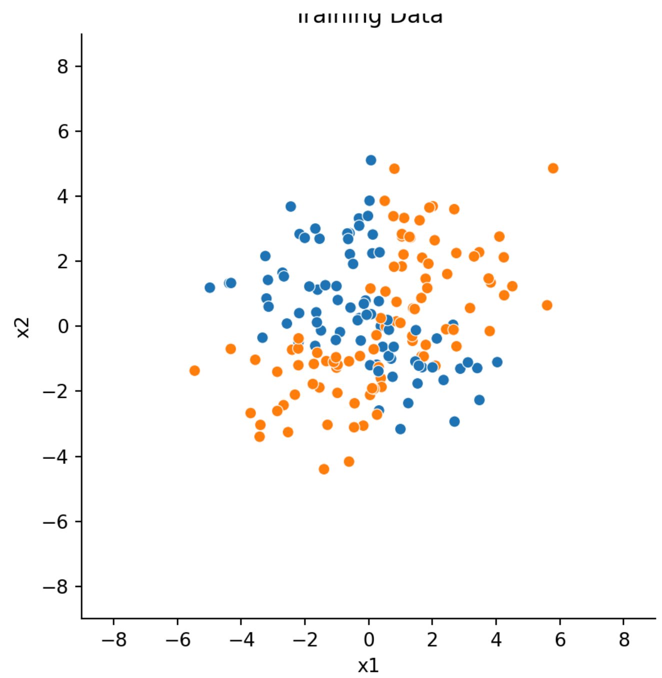


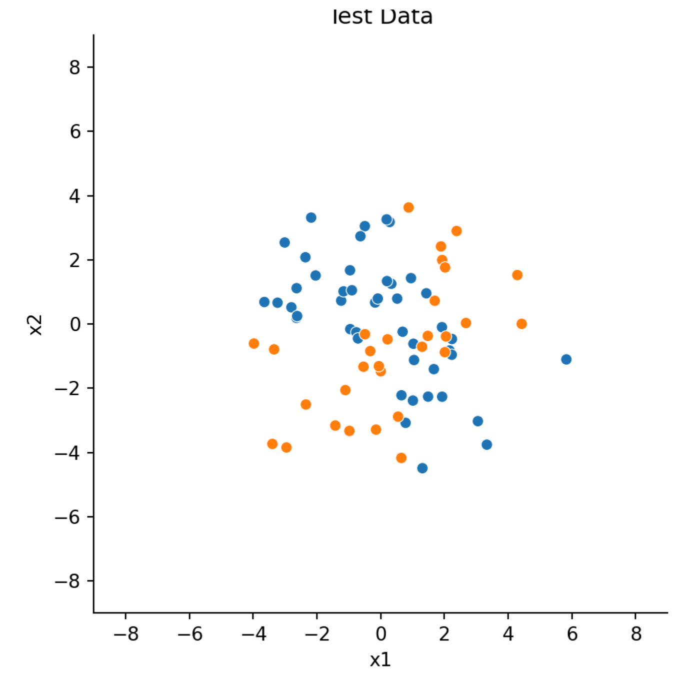
Test-train split

```
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, random_state=1234
)
```


Training Data
x2
x1


Test Data
x2
x1


x2
x1


x2
x1

Model

```
with pm.Model(coords = {"coeffs": X_lab}) as model:
    # data containers
    X = pm.Data("X", X_train)
    y = pm.Data("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)
```

Visualizing models

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

Fitting

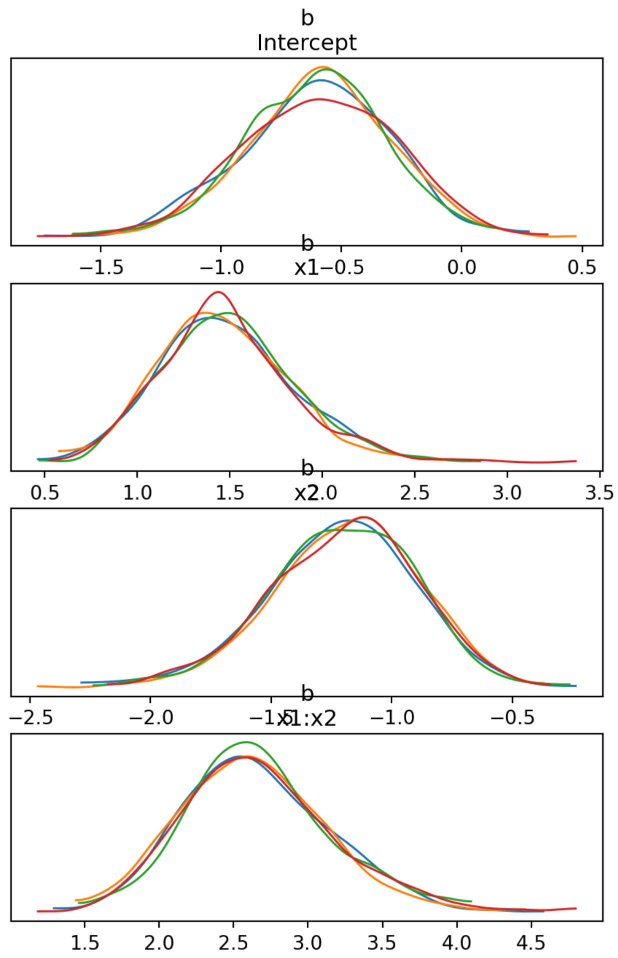
```
1  with model:
2      post = pm.sample(progressbar=False, random_seed=1234)
```

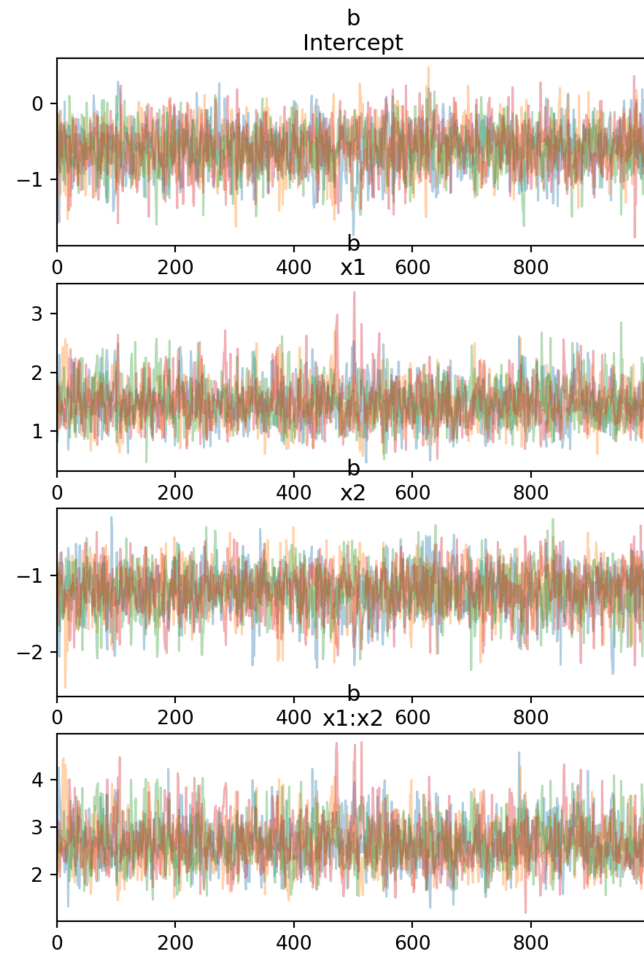
```
1  az.summary(post)
```

```
                 mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r
b[Intercept]   -0.600  0.307  -1.194   -0.050      0.007    0.005    2127.0    2351.0
b[x1]           1.477  0.358   0.831    2.170      0.008    0.006    1849.0    2268.0
b[x2]          -1.201  0.304  -1.774   -0.637      0.007    0.005    1910.0    2048.0
b[x1:x2]        2.658  0.500   1.800    3.688      0.013    0.010    1632.0    2144.0
p[0]            0.224  0.074   0.084    0.354      0.002    0.001    1912.0    2306.0
...               ...    ...     ...      ...        ...      ...       ...       ...
p[170]          0.404  0.089   0.234    0.566      0.002    0.001    2496.0    2485.0
p[171]          0.014  0.019   0.000    0.043      0.000    0.001    1739.0    1815.0
p[172]          0.985  0.015   0.959    1.000      0.000    0.001    2046.0    1982.0
p[173]          0.545  0.067   0.414    0.660      0.001    0.001    3491.0    3288.0
p[174]          1.000  0.000   1.000    1.000      0.000    0.000    1702.0    2679.0

[179 rows x 9 columns]
```

Trace plots

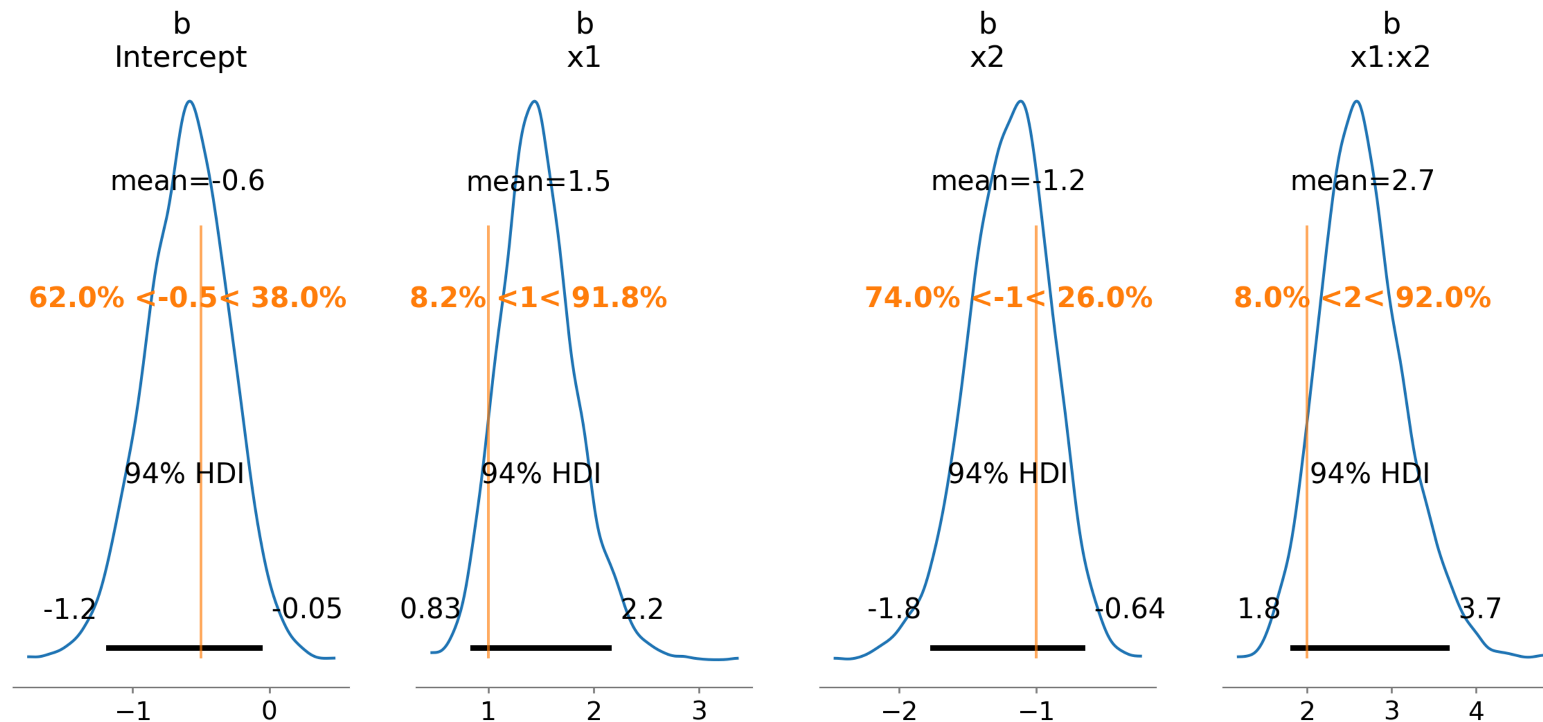
```
axs = az.plot_trace(post, var_names="b", compact=False)
plt.show()
```


b
Intercept
−1.5
−1.0
−0.5
0.0
0.5
b
x1
0.5
1.0
1.5
2.0
2.5
3.0
3.5
b
x2
−2.5
−2.0
−1.5
−1.0
−0.5
b
x1:x2
1.5
2.0
2.5
3.0
3.5
4.0
4.5


b
Intercept
0
−1
0
200
400
600
800
b
x1
3
2
1
b
x2
−1
−2
b
x1:x2
4
3
2

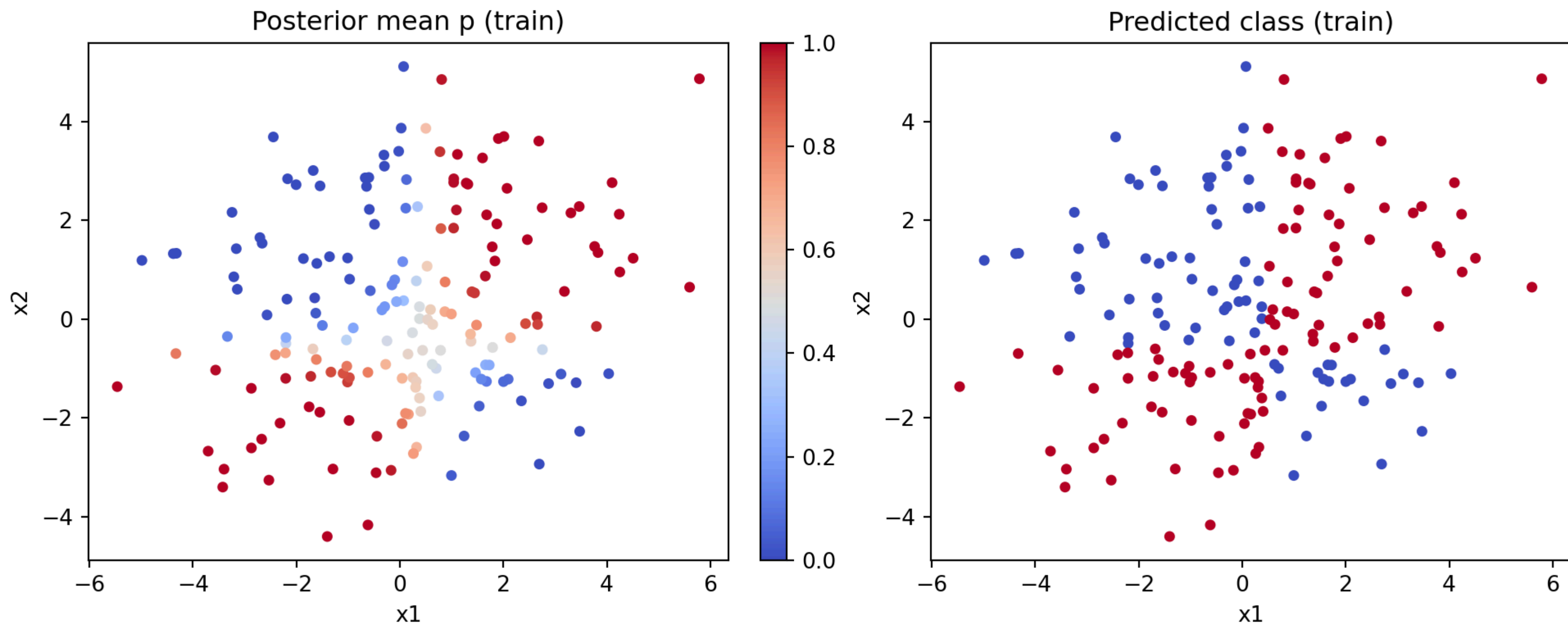
Posterior plots

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



Posterior samples

```
1  p_post = post.posterior["p"].mean(dim=["chain", "draw"])
2  fig, axes = plt.subplots(1, 2, figsize=(10, 4), layout="constrained")
3
4  sc = axes[0].scatter(X_train[:,1], X_train[:,2], c=p_post, cmap='coolwarm', vmin=0, vmax=1, s=15)
5  axes[0].set(xlabel="x1", ylabel="x2", title="Posterior mean p (train)")
6  cb = plt.colorbar(sc, ax=axes[0])
7
8  axes[1].scatter(X_train[:,1], X_train[:,2], c=(p_post > 0.5).astype(int), cmap='coolwarm', vmin=0, vmax
9  axes[1].set(xlabel="x1", ylabel="x2", title="Predicted class (train)")
10 plt.show()
```



Out-of-sample predictions

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

```
1 post
```

arviz.InferenceData

- ▶ posterior
- ▶ sample_stats
- ▶ observed_data
- ▶ constant_data

Posterior predictive summary

```
1  az.summary(
2    post.posterior_predictive
3  )
```

```
         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]  0.017  0.128   0.000    0.000      0.002    0.008    3965.0    3965.0    1.0
obs[2]  0.558  0.497   0.000    1.000      0.008    0.001    3813.0    3813.0    1.0
obs[3]  0.000  0.000   0.000    0.000      0.000      NaN    4000.0    4000.0    NaN
obs[4]  0.245  0.430   0.000    1.000      0.007    0.004    3894.0    3894.0    1.0
...       ...    ...     ...      ...        ...      ...       ...       ...    ...
p[70]   0.181  0.138   0.007    0.444      0.002    0.002    3667.0    3370.0    1.0
p[71]   0.961  0.033   0.901    0.999      0.001    0.001    2179.0    2413.0    1.0
p[72]   0.000  0.000   0.000    0.000      0.000    0.000    1551.0    1793.0    1.0
p[73]   1.000  0.000   1.000    1.000      0.000    0.000    1786.0    1980.0    1.0
p[74]   0.676  0.102   0.486    0.865      0.002    0.001    3477.0    3181.0    1.0

[150 rows x 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"])
```

```
1 print(p_train)
```

```
<xarray.DataArray 'p' (p_dim_0: 175)> Size: 1kB
array([0.22354, 0.57537, 1.     , 0.00002, 0.962
       0.99993, 0.51717, 1.     , 0.99992, 0.962
       1.     , 1.     , 0.91063, 0.59602, 0.014
       0.01492, 0.99939, 0.74472, 0.99996, 0.977
       0.81544, 0.99442, 0.14779, 0.     , 0.999
       0.93578, 0.47611, 0.008  , 0.33411, 0.071
       0.67693, 0.01106, 0.0376 , 1.     , 0.
       0.89679, 0.18902, 0.     , 0.11247, 0.577
       0.0004 , 0.05597, 0.     , 0.56356, 0.074
       1.     , 0.04337, 0.99839, 0.84047, 0.000
       0.93892, 0.25884, 0.55489, 0.11686, 0.000
       1.     , 0.24473, 0.00055, 0.82414, 0.001
       0.7419 , 0.99982, 0.09778, 0.06814, 0.000
       0.9969 , 0.03411, 0.1611 , 0.43356, 0.007
       0.69623, 0.99907, 0.99812, 1.     , 0.010
       0.00432, 0.96826, 0.33577, 0.00199, 0.584
       0.01221, 1.     , 0.66046, 0.38862, 0.884
       0.77121, 0.59311, 0.00975, 0.99965, 0.139
       0.99824, 0.     , 0.     , 0.98214, 0.242
       0.6764 , 0.99849, 0.77462, 0.99031, 0.462
       0.99998, 0.47801, 0.92261, 0.14541, 1.
```

```
1 print(p_test)
```

```
<xarray.DataArray 'p' (p_dim_0: 75)> Size: 600B
array([1.     , 0.01609, 0.57241, 0.     , 0.253
       0.31829, 0.00197, 0.50455, 0.99644, 0.205
       0.99991, 0.75426, 0.00075, 0.65654, 0.894
       0.56451, 0.00047, 0.97465, 0.63495, 0.000
       0.00002, 0.85128, 0.00041, 0.38313, 0.
       1.     , 0.13034, 0.0919 , 0.00002, 0.082
       0.00486, 0.     , 0.15431, 0.13207, 0.640
       0.24031, 0.0774 , 0.00348, 0.74317, 0.143
       0.9803 , 0.9225 , 0.23294, 0.37334, 0.000
       0.00006, 1.     , 0.67627])
Coordinates:
  * p_dim_0  (p_dim_0) int64 600B 0 1 2 3 4 5 6
```

ROC & AUC

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

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
```

```
0.9377777777777778
```

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
```

```
0.9619736842105263
```

ROC Curves

```python
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()
```