Lec 19 - PyMC3 + ArviZ

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

pymc3 + ArviZ

> PyMC3 is a probabilistic programming package for Python that allows users to fit Bayesian models using a variety of numerical methods, most notably Markov chain Monte Carlo (MCMC) and variational inference (VI). Its flexibility and extensibility make it applicable to a large suite of problems. Along with core model specification and fitting functionality, PyMC3 includes functionality for summarizing output and for model diagnostics.
>
> ArviZ is a Python package for exploratory analysis of Bayesian models. Includes functions for posterior analysis, data storage, sample diagnostics, model checking, and comparison. The goal is to provide backend-agnostic tools for diagnostics and visualizations of Bayesian inference in Python, by first converting inference data into xarray objects.

```
import pymc3 as pm
import arviz as az
```

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)
```

```
## TypeError: No model on context stack, which is needed to instantiate distributions. Add variable inside a 'w
```

Additional components can be added to an existing model via additional `with` statements (only the first needs `pm.Model()`)

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

```
norm.vars
```

Random Variables

`pm.Normal()` is an example of a PyMC distribution, which are used to construct models, these are implemented using the `FreeRV` class which is used for all of the builtin distributions (and can be used to create custom distributions). Some useful methods and attributes,

```
norm.x.dshape
```

```
## ()
```

```
norm.x.dsize
```

```
## 1
```

```
norm.x.distribution
```

```
## <pymc3.distributions.continuous.Normal object a
```

```
norm.x.init_value
```

```
## array(0.)
```

```
norm.model
```

```
norm.x.random()
```

```
## array(0.94831)
```

```
norm.y.random()
```

```
## array([ 0.65532,  0.0627 , -0.94107])
```

```
norm.x.logp({"x": 0, "y": [0,0,0]})
```

```
## array(-0.91894)
```

```
norm.y.logp({"x": 0, "y": [0,0,0]})
```

```
## array(-2.75682)
```

```
norm.logp({"x": 0, "y": [0,0,0]})
```

Variable heirarchy

Note that we defined $y | x \sim \mathrm{N}(x, 1)$, so what is happening when we use `norm.y.random()`?

```
norm.y.random()
```

```
## array([-0.05382,  0.27083,  1.12145])
```

```
obs = norm.y.random(size=1000)
np.mean(obs)
```

```
## 0.017603505100856152
```

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

```
## 2.0796380254179776
```

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

```
## 1.442095012618093
```

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.

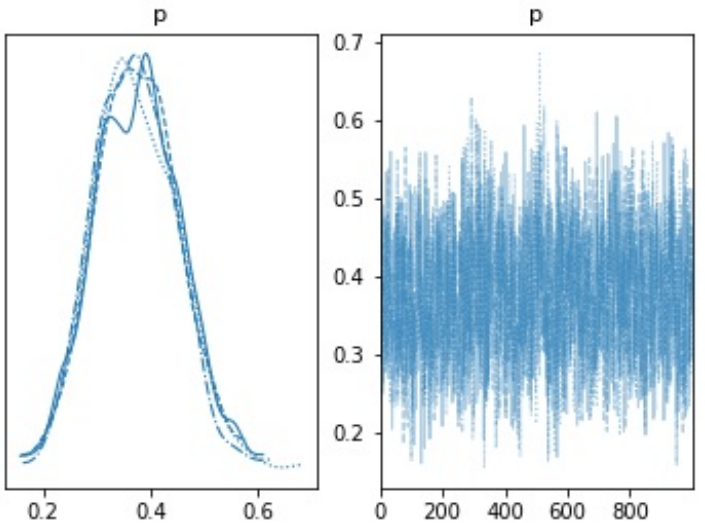
```
with pm.Model() as beta_binom:
  p = pm.Beta("p", alpha=10, beta=10)
  x = pm.Binomial("x", n=20, p=p, observed=5)
```

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

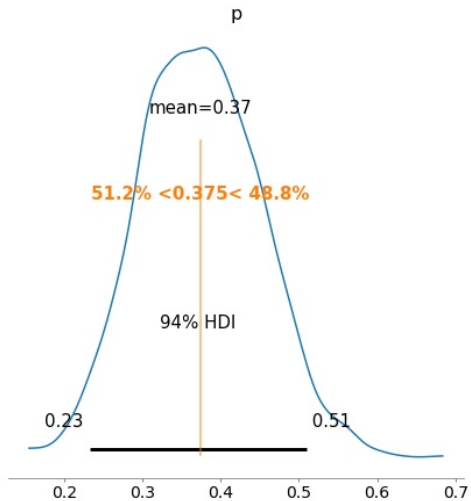
```
with beta_binom:
  trace = pm.sample(return_inferencedata=True, random_seed=1234)
```

```
##
## 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 6 seconds.
```

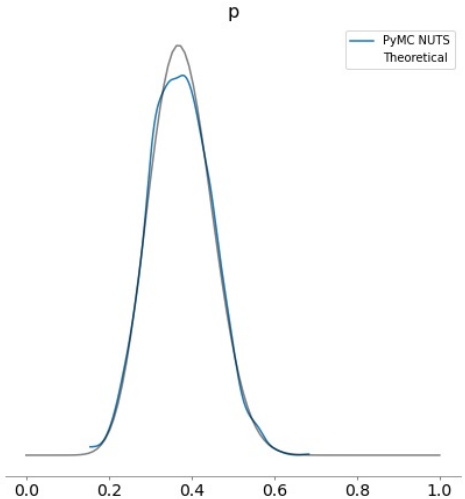
```
ax = az.plot_trace(trace, figsize=(6,4))
plt.show()
```



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



```
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()
```



InferenceData results

```
print(trace)
```

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

```
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

```
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 0.4051 0.4491 0.4985 ... 0.353 0.3691 0.3691
## Attributes:
##     created_at:                 2022-03-18T17:47:37.298366
##     arviz_version:              0.11.4
##     inference_library:          pymc3
##     inference_library_version:  3.11.5
##     sampling_time:              5.83094596862793
##     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)
```

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     0.405115
##       1     0.449149
##       2     0.498481
##       3     0.522682
##       4     0.346336
## ...               ...
## 3     995   0.380507
##       996   0.404883
##       997   0.353017
##       998   0.369109
##       999   0.369109
##
## [4000 rows x 1 columns]
```

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

```
##       chain         p
## draw
## 0         0  0.405115
## 1         0  0.449149
## 2         0  0.498481
## 3         0  0.522682
## 4         0  0.346336
## ...     ...       ...
## 995       0  0.463122
## 996       0  0.437503
## 997       0  0.437503
## 998       0  0.339669
## 999       0  0.393476
##
## [1000 rows x 2 columns]
```

MultiTrace results

```
with beta_binom:
  mt = pm.sample(random_seed=1234)
```

```
## 
## FutureWarning: In v4.0, pm.sample will return an `arviz.InferenceData` object instead of a `MultiTrace` by d
   You can pass return_inferencedata=True or return_inferencedata=False to be safe and silence this warning.
##   return wrapped_(*args_, **kwargs_)
## 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 5 seconds.
```

```
mt
```

```
## <MultiTrace: 4 chains, 1000 iterations, 2 variables>
```

```
type(mt)
```

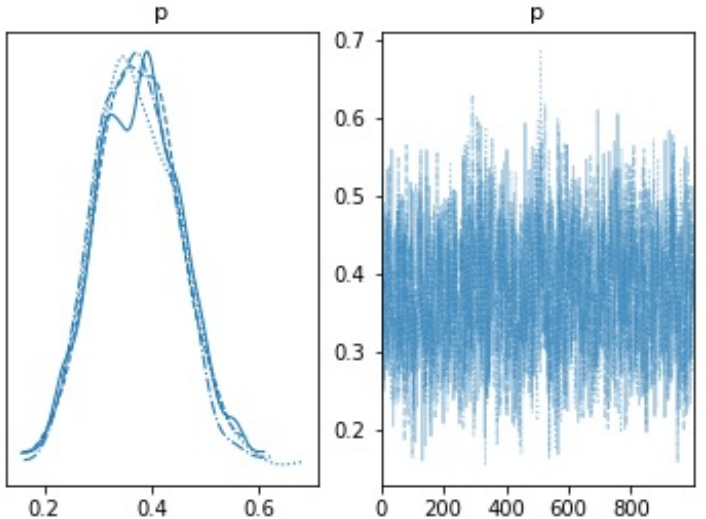
```
## <class 'pymc3.backends.base.MultiTrace'>
```

```
ax = az.plot_trace(mt, figsize=(6,4))
```

```
## Got error No model on context stack. trying to find log_likelihood in translation.
## FutureWarning: Using `from_pymc3` without the model will be deprecated in a future release. Not using the mo
   return less accurate and less useful results. Make sure you use the model argument or call from_pymc3 within
   context.
##   warnings.warn(
## Got error No model on context stack. trying to find log_likelihood in translation.
```

```
plt.show()
```

```
with beta_binom:
  ax = az.plot_trace(mt, figsize=(6,4))
plt.show()
```



Working with MultiTrace

```
mt['p']
```

```
## array([0.40512, 0.44915, 0.49848, 0.52268, 0.34634, 0.33228, 0.2552 , 0.34267, 0.24986, 0.38481, 0.39414, 0.
   0.4012 , 0.42459, 0.3994 , 0.38614, 0.49096, 0.4057 , 0.40255, 0.43986, 0.40974,
##        0.53249, 0.44479, 0.29335, 0.48019, 0.41937, 0.38748, 0.37968, 0.26982, 0.27831, ..., 0.46317, 0.5039
   0.37687, 0.25664, 0.34679, 0.53207, 0.56233, 0.39998, 0.39998, 0.35253, 0.22216,
##        0.54898, 0.44927, 0.48805, 0.27699, 0.27699, 0.27699, 0.30037, 0.23423, 0.27166, 0.31804, 0.24879, 0.
   0.36096, 0.367  , 0.38051, 0.40488, 0.35302, 0.36911, 0.36911])
```

```
mt['p'].shape
```

```
## (4000,)
```

```
mt['p', 500:].shape
```

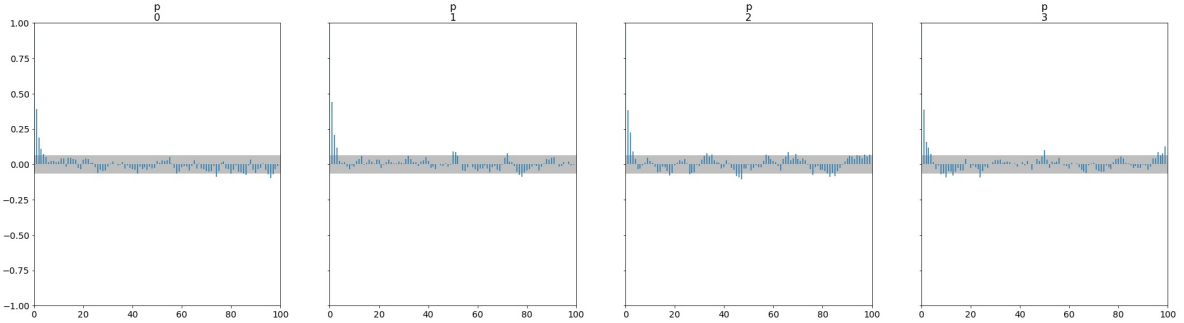
```
## (2000,)
```

```
mt.get_values(varname="p", burn=500, thin=10, chains=[0,1]).shape
```

```
## (100,)
```

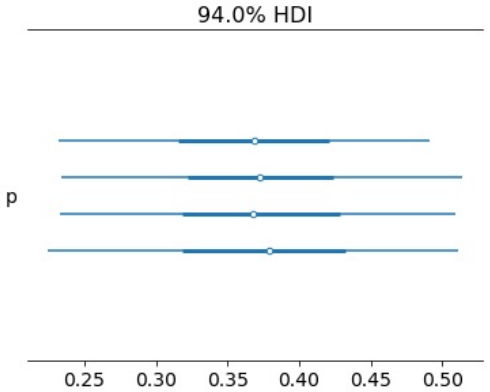
Autocorrelation plots

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



Forrst plots

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



Other useful diagnostics

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

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

```
##     mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
## p  0.374  0.076   0.232    0.509      0.002    0.001    1596.0    2654.0    1.0
```

individual methods are available for each statistics,

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

```
## <xarray.Dataset>
## Dimensions:  ()
## Data variables:
##     p        float64 1.596e+03
```

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

```
## <xarray.Dataset>
## Dimensions:  ()
## Data variables:
##     p        float64 1.001
```

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

```
## <xarray.Dataset>
## Dimensions:  ()
## Data variables:
##     p        float64 2.654e+03
```

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

```
## <xarray.Dataset>
## Dimensions:  ()
## Data variables:
##     p        float64 0.001905
```

Demo 1 - Linear regression

Given the below data, we will 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

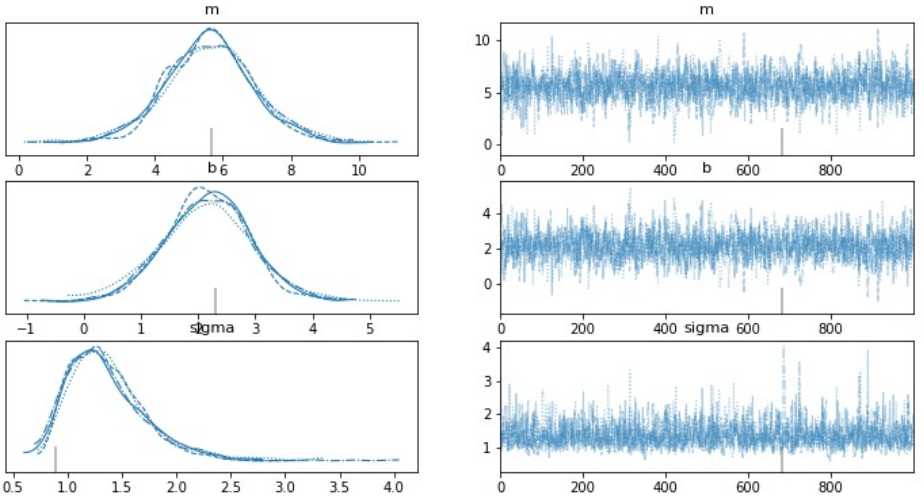
```
with pm.Model() as lm:
  m = pm.Normal('m', mu=0, sd=50)
  b = pm.Normal('b', mu=0, sd=50)
  sigma = pm.HalfNormal('sigma', sd=5)

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

  trace = pm.sample(return_inferencedata=True, random_seed=1234)
```

```
## 
## Auto-assigning NUTS sampler...
## Initializing NUTS using jitter+adapt_diag...
## Multiprocess sampling (4 chains in 4 jobs)
## NUTS: [sigma, b, m]
## Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 6 seconds.
## There was 1 divergence after tuning. Increase `target_accept` or reparameterize.
```

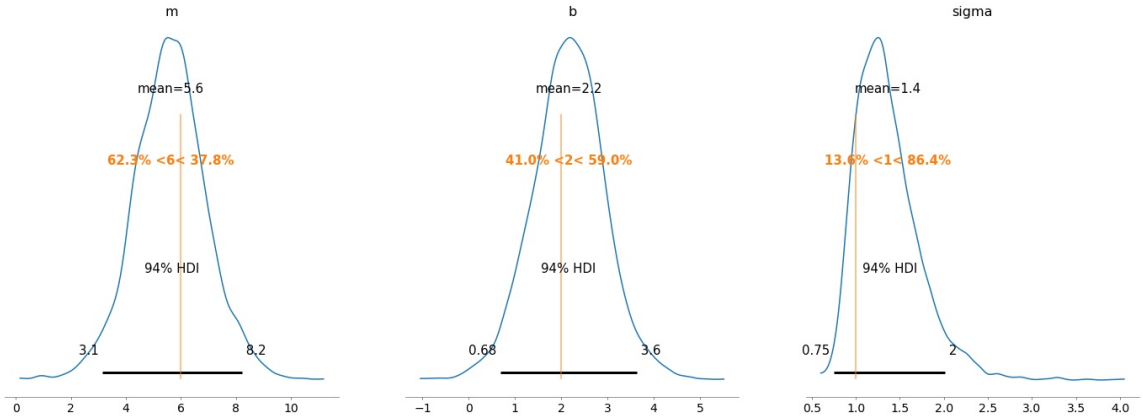
```
az.summary(trace)
```

```
##         mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
## m      5.620  1.327   3.145    8.235      0.037    0.026    1325.0    1519.0    1.0
## b      2.157  0.783   0.682    3.638      0.022    0.016    1268.0    1392.0    1.0
## sigma  1.363  0.369   0.754    2.013      0.009    0.007    1440.0    1461.0    1.0
```

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



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



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

post_m = trace.posterior['m'][0, -500:]
post_b = trace.posterior['b'][0, -500:]

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

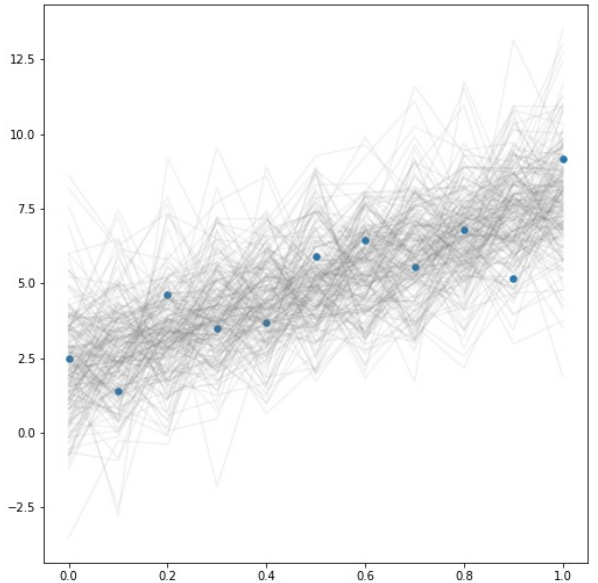
```
with lm:
  pp = pm.sample_posterior_predictive(trace, samples=200)
```

```
##
## UserWarning: samples parameter is smaller than nchains times ndraws, some draws and/or chains may not be rep
   in the returned posterior predictive sample
##   warnings.warn(
```

```
pp['y'].shape
```

```
## (200, 11)
```

```
plt.figure(layout="constrained")
plt.plot(x, pp['y'].T, c="grey", alpha=0.1)
plt.scatter(x, y, s=30, label='data')
plt.show()
```

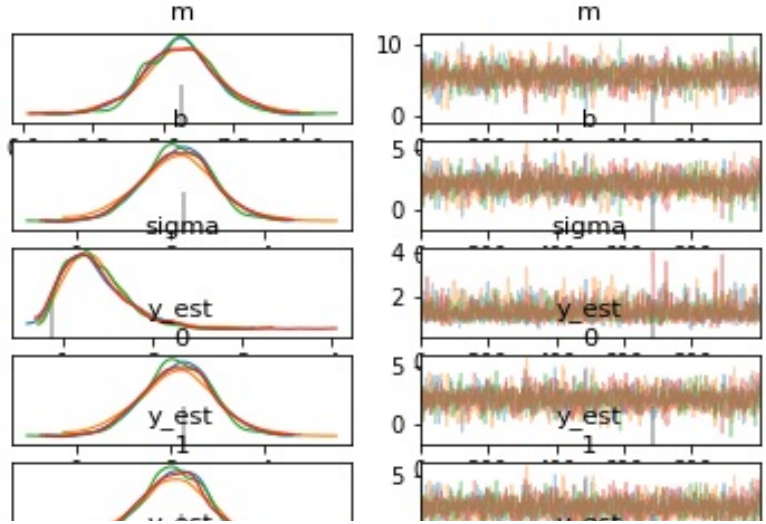


Model revision

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

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

  likelihood = pm.Normal('y', mu=y_est, sd=sigma, observed=y)

  trace = pm.sample(return_inferencedata=True, random_seed=1234)
  pp = pm.sample_posterior_predictive(trace, var_names=["y_est"], samples=200)
```

```
## ■■
## Auto-assigning NUTS sampler...
## Initializing NUTS using jitter+adapt_diag...
## Multiprocess sampling (4 chains in 4 jobs)
## NUTS: [sigma, b, m]
## Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 6 seconds.
## There was 1 divergence after tuning. Increase `target_accept` or reparameterize.
## UserWarning: samples parameter is smaller than nchains times ndraws, some draws and/or chains may not be rep
   in the returned posterior predictive sample
##   warnings.warn(
```

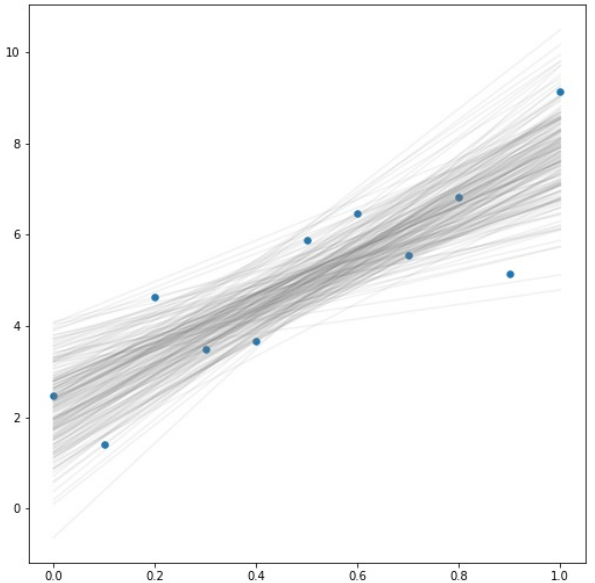
```
plt.figure(layout="constrained")
ax = az.plot_trace(trace, compact=False, figsize=(6,12))
plt.show()
```



```
pp['y_est'].shape
```

```
## (200, 11)
```

```
plt.figure(layout="constrained")
plt.plot(x, pp['y_est'].T, c="grey", alpha=0.1)
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_lasso:
  b = pm.Laplace("beta", 0, 1, shape=k)#lam*tau, shape=k)
  y_est = X @ b
  s = pm.HalfNormal('sigma', sd=1)

  likelihood = pm.Normal("y", mu=y_est, sigma=s, observed=y)

  trace = pm.sample(return_inferencedata=True, random_seed=1234)
```

```
## 
## Auto-assigning NUTS sampler...
## Initializing NUTS using jitter+adapt_diag...
## Multiprocess sampling (4 chains in 4 jobs)
## NUTS: [sigma, beta]
## Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 18 seconds.
## There were 2 divergences after tuning. Increase `target_accept` or reparameterize.
## The acceptance probability does not match the target. It is 0.878942077718847, but should be close to 0.8. T
   increase the number of tuning steps.
## The estimated number of effective samples is smaller than 200 for some parameters.
```

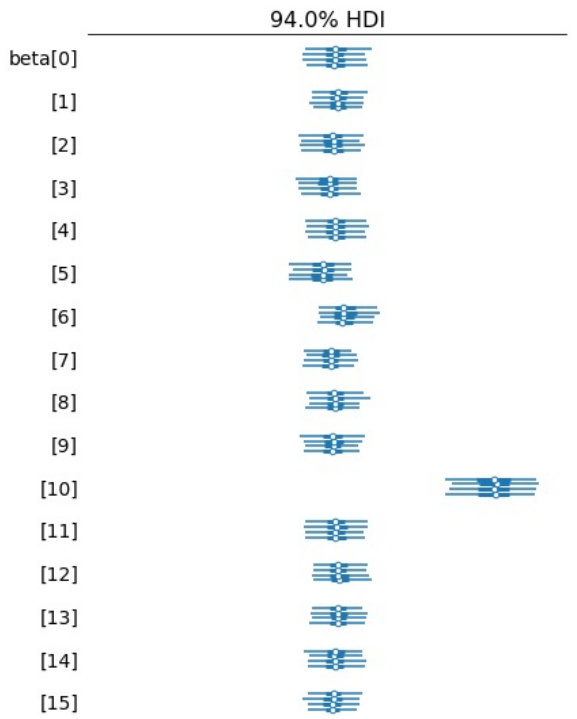
```
az.summary(trace)
```

```
##             mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
## beta[0]    0.067  0.861  -1.650    1.681      0.015    0.015    3234.0    1938.0   1.00
## beta[1]    0.215  0.729  -1.133    1.693      0.012    0.013    3632.0    2284.0   1.00
## beta[2]   -0.080  0.852  -1.789    1.501      0.014    0.015    3866.0    2652.0   1.00
## beta[3]   -0.290  0.814  -1.926    1.193      0.016    0.015    2870.0    1729.0   1.00
## beta[4]    0.079  0.809  -1.479    1.691      0.014    0.014    3577.0    2158.0   1.00
## ...          ...    ...     ...      ...        ...      ...       ...       ...    ...
## beta[96]   0.106  0.726  -1.271    1.542      0.013    0.013    3471.0    2487.0   1.00
## beta[97]  -0.156  0.716  -1.591    1.160      0.013    0.013    3188.0    1798.0   1.00
## beta[98]   0.289  0.763  -1.076    1.827      0.014    0.015    3107.0    2408.0   1.00
## beta[99]  -0.278  0.768  -1.747    1.205      0.013    0.013    3575.0    2568.0   1.00
## sigma      0.980  0.478   0.275    1.859      0.046    0.032     102.0     211.0   1.05
##
## [101 rows x 9 columns]
```

```
az.summary(trace).iloc[[0,10,20,30,40,50,60,70,80,100]]
```

```
##             mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
## beta[0]    0.067  0.861  -1.650    1.681      0.015    0.015    3234.0    1938.0   1.00
## beta[10]   8.327  1.242   5.945   10.622      0.027    0.019    2075.0    2710.0   1.00
## beta[20]  -8.288  1.335 -10.697   -5.733      0.030    0.021    2003.0    1746.0   1.00
## beta[30]   8.610  1.023   6.678   10.447      0.023    0.017    2011.0    1702.0   1.00
## beta[40]  -8.765  1.507 -11.485   -5.929      0.030    0.022    2461.0    2531.0   1.00
## beta[50]   8.966  1.016   6.995   10.860      0.023    0.016    2035.0    1842.0   1.00
## beta[60]  -9.248  1.121 -11.381   -7.162      0.022    0.015    2708.0    2371.0   1.00
```

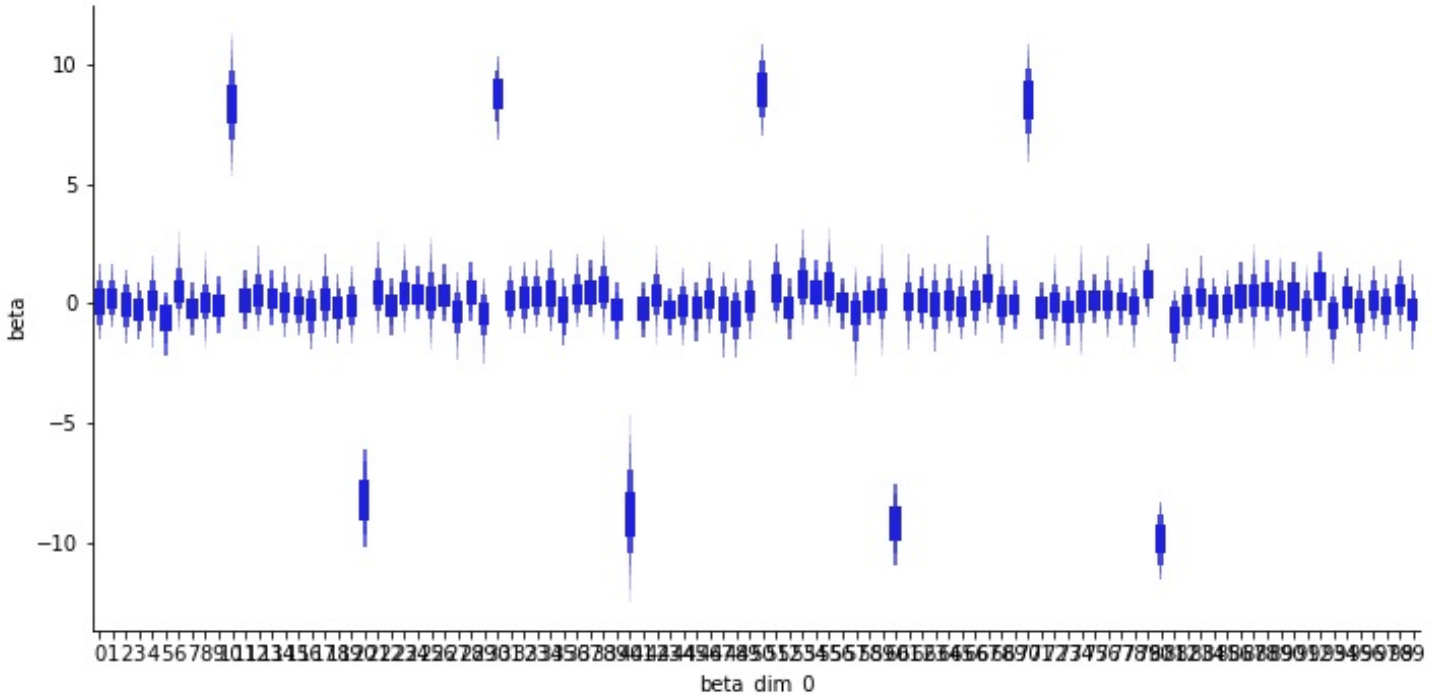
```
ax = az.plot_forest(trace)
plt.tight_layout()
plt.show()
```



Plot helper

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

  sns.catplot(x="beta_dim_0", y="beta", data=post, kind="boxen", linewidth=0, color='blue', aspect=2, sho
  plt.tight_layout()
  plt.show()
```

```
plot_slope(trace)
```

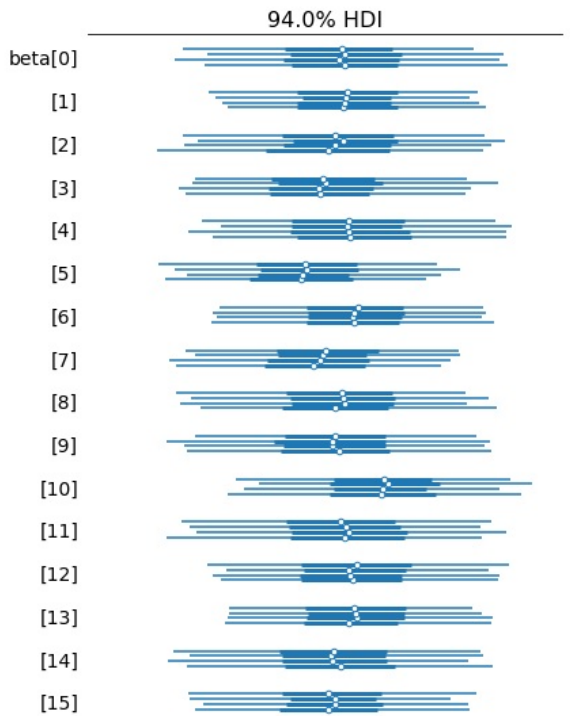


Weakly Informative Prior

```
with pm.Model() as bayes_weak:
  b = pm.Normal("beta", 0, 10, shape=k)
  y_est = X @ b

  s = pm.HalfNormal('sigma', sd=2)

  likelihood = pm.Normal("y", mu=y_est, sigma=s, observed=y)

  trace = pm.sample(return_inferencedata=True, random_seed=12345)
```

```
## 
## Auto-assigning NUTS sampler...
## Initializing NUTS using jitter+adapt_diag...
## Multiprocess sampling (4 chains in 4 jobs)
## NUTS: [sigma, beta]
## Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 55 seconds.
## The acceptance probability does not match the target. It is 0.9760397075294559, but should be close to 0.8. 
   increase the number of tuning steps.
## The chain reached the maximum tree depth. Increase max_treedepth, increase target_accept or reparameterize.
## There was 1 divergence after tuning. Increase `target_accept` or reparameterize.
## There were 15 divergences after tuning. Increase `target_accept` or reparameterize.
## The acceptance probability does not match the target. It is 0.7066410867916934, but should be close to 0.8. 
   increase the number of tuning steps.
## There was 1 divergence after tuning. Increase `target_accept` or reparameterize.
```

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



```
plot_slope(trace)
```