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.

```python
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(-0.72791)
```

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

```
## array([-0.55974, -0.36685, -1.39941])
```

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

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

```
## array([1.35141, 1.72697, 1.90376])
```

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

```
## -0.036509618618011114
```

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

```
## 1.9445967710025949
```

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

```
## 1.3944879960051986
```

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.

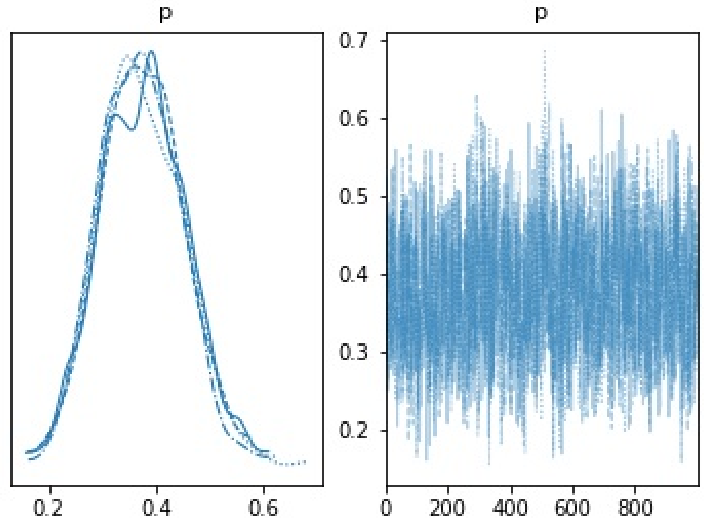
```
with pm.Model() as beta_binom:
  p = pm.Beta("p", alpha=10, beta=10)
  x = pm.Binomial("x", n=20, p=p, observed=5)
```

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

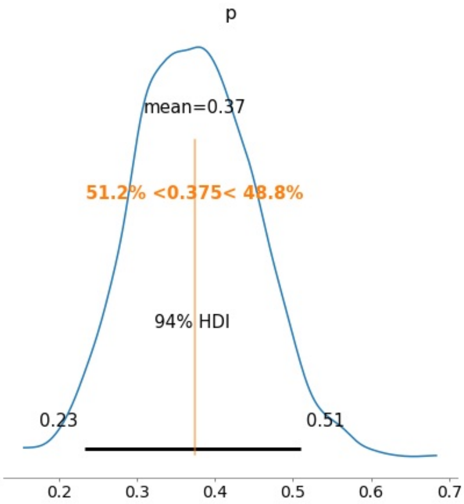
```
with beta_binom:
  trace = pm.sample(return_inferencedata=True, random_seed=1234)
```

```
## █
## 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()
```

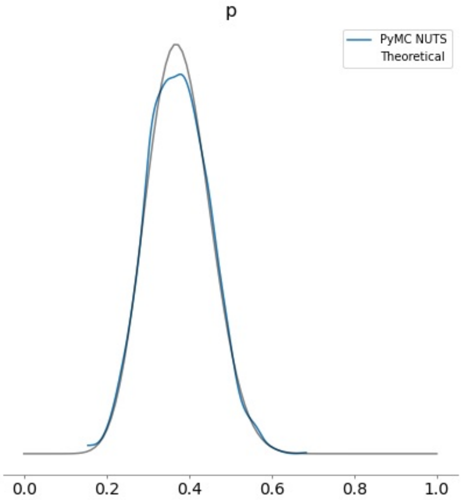


```
ax = az.plot_posterior(trace, ref_val=[15/40])
plt.show()
```



```
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

```python
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:                 2022-03-18T13:28:45.852102
##     arviz_version:              0.11.4
##     inference_library:          pymc3
##     inference_library_version:  3.11.5
##     sampling_time:              5.686646938323975
##     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)
```

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]
```

.pull-right[

```
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]
```