PyMC - Samplers

Lecture 24

Dr. Colin Rundel

Samplers - Metropolis-Hastings

Algorithm

For a parameter of interest start with an initial value θ_0 then for the nex sample $(t+1)$,

1. Generate a proposal value θ' from a proposal distribution $q(\theta'|\theta_t)$.
2. Calculate the acceptance probability,

$$\alpha = \min\left(1, \frac{P(\theta'|x)}{P(\theta_t|x)}\frac{q(\theta_t|\theta')}{q(\theta'|\theta_t)}\right)$$

   where $P(\theta|x)$ is the target posterior distribution.
3. Accept proposal θ' with probability α, if accepted $\theta_{t+1} = \theta'$ else $\theta_{t+1} = \theta_t$.

Some considerations:

- Choice of the proposal distribution matters a lot
- Results are for the limit as $t \longrightarrow \infty$
- Concerns are around computational efficiency

Banana Distribution

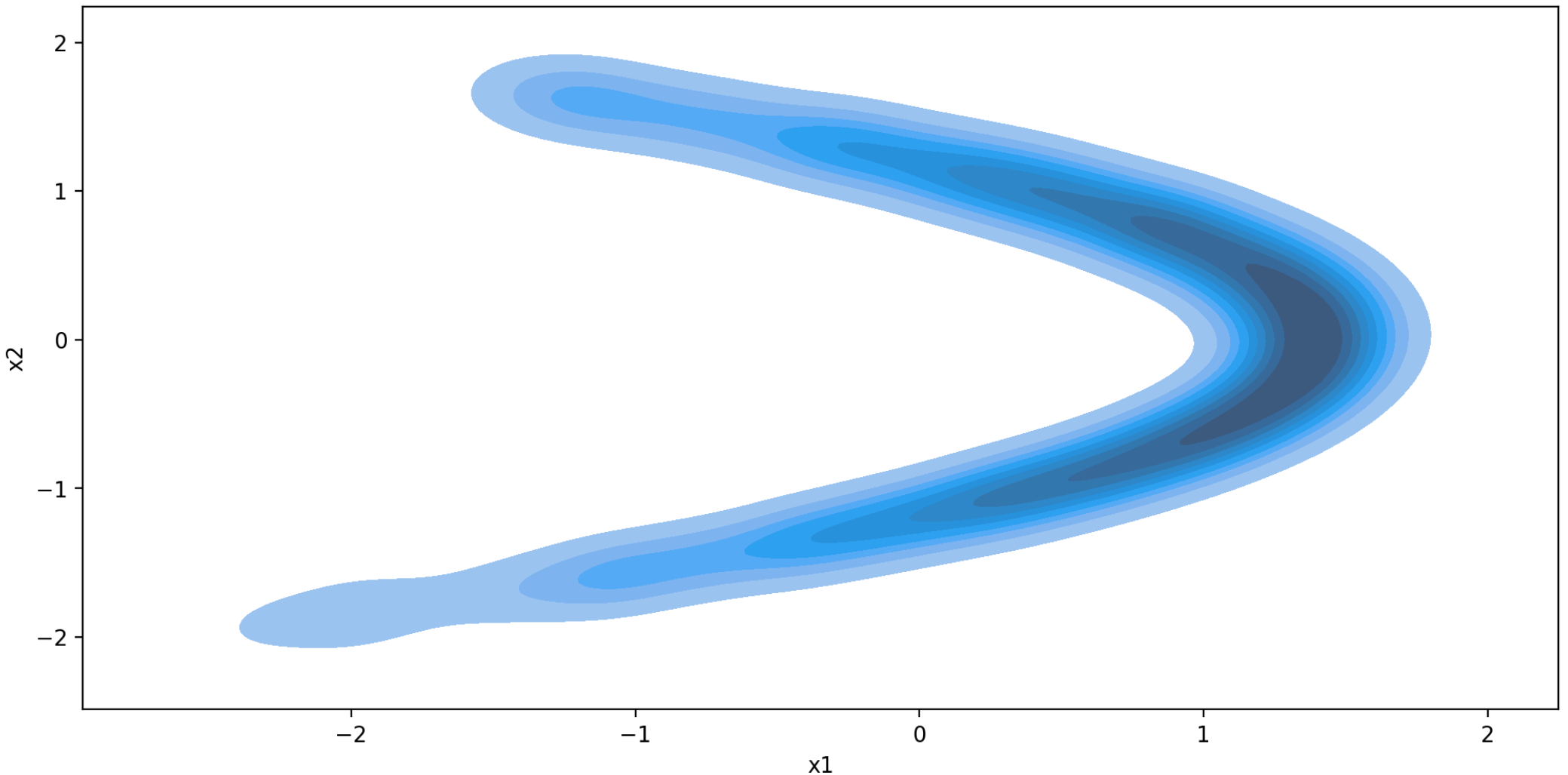
```python
# Data
n = 100
x1_mu = .75
x2_mu = .75
y = pm.draw(pm.Normal.dist(mu=x1_mu+x2_mu**2, sigma=1, shape=n))

# Model
with pm.Model() as banana:
  x1 = pm.Normal("x1", mu=0, sigma=1)
  x2 = pm.Normal("x2", mu=0, sigma=1)

  y = pm.Normal("y", mu=x1+x2**2, sigma=1, observed=y)

  trace = pm.sample(draws=50000, chains=1, random_seed=1234)
```

```
Initializing NUTS using jitter+adapt_diag...
Sequential sampling (1 chains in 1 job)
NUTS: [x1, x2]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elap
━━━━━━━━━━━━━━━━━━━━━━━━	51000	1606	0.11	13	3672.13 draws/s	0:00

```
Sampling 1 chain for 1_000 tune and 50_000 draw iterations (1_000 + 50_000 draws total) took 14 se
There were 1606 divergences after tuning. Increase `target_accept` or reparameterize.
Only one chain was sampled, this makes it impossible to run some convergence checks
```

Joint posterior of x1 & x2



Metropolis-Hastings Sampler

Model | Summary | Traces | Trajectories | ACF

```
1 with banana:
2   mh = pm.sample(
3     draws=100, tune=0,
4     step=pm.Metropolis([x1,x2]),
5     random_seed=1234
6   )
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [x1]
>Metropolis: [x2]
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling S
———	100	False	1.00	0.44	7329.33 dr
———	100	False	1.00	0.00	5711.69 dr
———	100	False	1.00	0.09	4940.11 dr
———	100	False	1.00	4.26	3876.04 dr

```
Sampling 4 chains for 0 tune and 100 draw iterations (0 + 400 draws total) took 0 seco
The rhat statistic is larger than 1.01 for some parameters. This indicates problems du
```

MH with Tuning

Model | Summary | Traces | Trajectories | ACF

```
with banana:
  mht = pm.sample(
    draws=100, tune=1000,
    step=pm.Metropolis([x1,x2]),
    random_seed=1234
  )
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [x1]
>Metropolis: [x2]
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elap
▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬	1100	False	0.27	0.34	12710.46 draws/s	0:00
▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬	1100	False	0.53	0.65	10839.79 draws/s	0:00
▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬	1100	False	0.90	0.00	9441.48 draws/s	0:00
▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬	1100	False	0.43	0.38	7260.90 draws/s	0:00

```
Sampling 4 chains for 1_000 tune and 100 draw iterations (4_000 + 400 draws total) took 0 seconds.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during sampli
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is r
```

Effects of tuning / burn-in

There are two confounded effects from letting the sampler tune / burn-in:

1. We have let the sampler run for 1000 iterations - this gives it a chance to find the area's of higher density and settle in.

   This also makes each chain less sensitive to their initial starting position.

2. We have also tuned the size of the MH proposals to achieve a better acceptance rates - this lets the chains better explore the target distribution.

More samples?

Model | Summary | Traces | Trajectories | ACF

```
1 with banana:
2   mh_more = pm.sample(
3     draws=1000, tune=1000,
4     step=pm.Metropolis([x1,x2]),
5     random_seed=1234
6   )
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [x1]
>Metropolis: [x2]
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elap
	2000	False	0.27	0.00	14241.36 draws/s	0:00
	2000	False	0.53	0.00	13568.00 draws/s	0:00
	2000	False	0.90	0.15	12713.09 draws/s	0:00
	2000	False	0.43	0.74	8478.92 draws/s	0:00

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 0 secc
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during samplir
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is r
```

Even more samples?

Model | Summary | Traces | Trajectories | ACF

```
1 with banana:
2   mh_more2 = pm.sample(
3     draws=10000, tune=1000,
4     step=pm.Metropolis([x1,x2]),
5     random_seed=1234
6   )
7
8 mh_more_thin = mh_more2.sel(draw=slice(0,None,10))
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [x1]
>Metropolis: [x2]
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elap
————————————	11000	False	0.27	0.36	15034.54 draws/s	0:00
————————————	11000	False	0.53	0.00	14826.28 draws/s	0:00
————————————	11000	False	0.90	0.11	13806.51 draws/s	0:00
————————————	11000	False	0.43	0.79	8818.57 draws/s	0:00

```
Sampling 4 chains for 1_000 tune and 10_000 draw iterations (4_000 + 40_000 draws total) took 1 se
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during samplir
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is r
```

Bivariate Normal Distribution

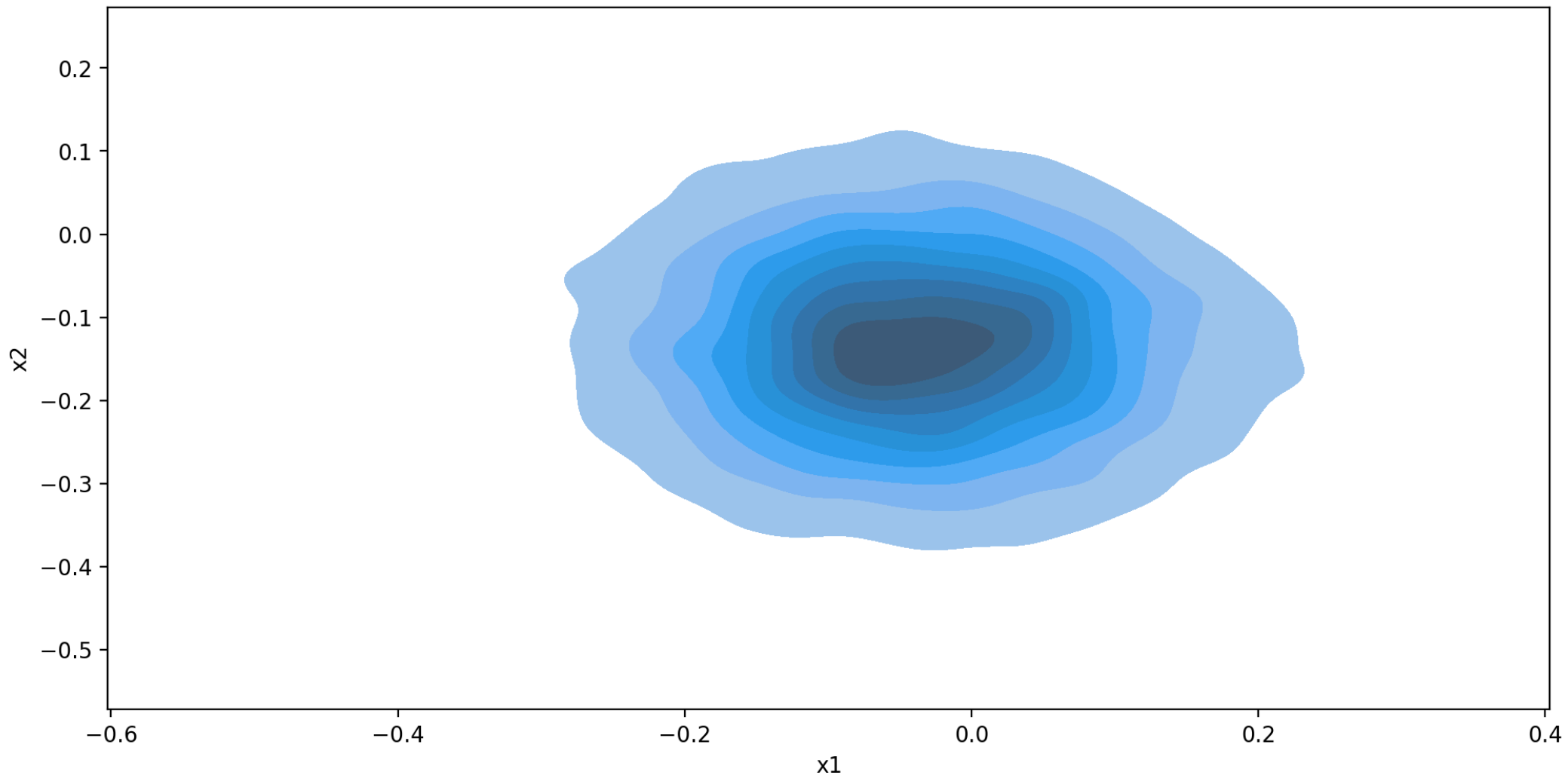
```
# Data
n = 100
y = pm.draw(pm.MvNormal.dist(mu=np.zeros(2), cov=np.eye(2,2), shape=(n,2)))

# Model
with pm.Model() as bv_normal:
  x1 = pm.Normal("x1", mu=0, sigma=1)
  x2 = pm.Normal("x2", mu=0, sigma=1)

  y = pm.MvNormal("y", mu=[x1,x2], cov=np.eye(2,2), observed=y)

  bv_trace = pm.sample(draws=10000, chains=1, random_seed=1234)
```

```
Initializing NUTS using jitter+adapt_diag...
Sequential sampling (1 chains in 1 job)
NUTS: [x1, x2]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elap
━━━━━━━━━━━━━━━━━━━━	11000	0	1.18	1	5878.70 draws/s	0:00

```
Sampling 1 chain for 1_000 tune and 10_000 draw iterations (1_000 + 10_000 draws total) took 2 sec
Only one chain was sampled, this makes it impossible to run some convergence checks
```

Joint posterior



BVM w/ MH

Model | Summary | Traces | Trajectories | ACF

```
with bv_normal:
  mh_bvn = pm.sample(
    draws=1000, tune=1000,
    step=pm.Metropolis([x1,x2]),
    random_seed=1234, cores=1
  )
```

```
Sequential sampling (2 chains in 1 job)
CompoundStep
>Metropolis: [x1]
>Metropolis: [x2]
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elap
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━	2000	False	0.53	0.00	9565.32 draws/s	0:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━	2000	False	0.59	0.00	4730.18 draws/s	0:00

```
Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) took 0 secc
We recommend running at least 4 chains for robust computation of convergence diagnostics
```

Sampler - Hamiltonian Methods

Background

Takes advantage of techniques developed in classical mechanics by imagining our parameters of interest as particles with a position and momentum,

$$H(\theta, \rho) = \underbrace{-\log p(\theta)}_{\text{potential}} \underbrace{- \log p(\rho|\theta)}_{\text{kinetic}}$$

Hamilton's equations of motion state give a set of partial differential equations governing the motion of the "particle".

A numerical integration method known as Leapfrog is then used to evolve the system some number of discrete steps forward in time.

Due to the approximate nature of the leapfrog integrator, a Metropolis acceptance step is typically used,

$$\alpha = \min\left(1, \exp\left(H(\theta, \rho) - H(\theta', \rho')\right)\right)$$

Algorithm parameters

There are a couple of important tuning parameters that are used by Hamiltonian monte carlo methods:

- ϵ is the size of the discrete time steps
- M is the mass matrix (or metric) that is used to determine the kinetic energy from the momentum (ρ)
- L is the number of leapfrog steps to take per iteration

Generally most of these will be tuned automatically for you by your sampler of choice.

HamiltonianMC

Model | Summary | Traces | Trajectories | ACF

```
with banana:
  hmc = pm.sample(
    draws=1000, tune=1000,
    step=pm.HamiltonianMC([x1,x2]),
    random_seed=1234
  )
```

```
Multiprocess sampling (4 chains in 4 jobs)
HamiltonianMC: [x1, x2]
```

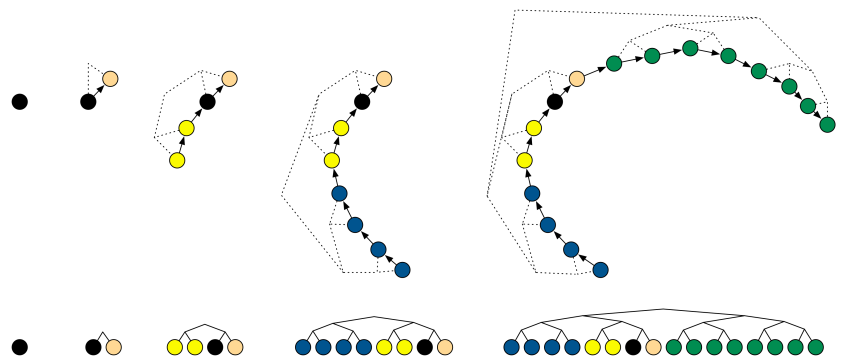
Progress	Draws	Sampling Speed	Elapsed	Remaining
	2000	4950.57 draws/s	0:00:00	0:00:00
	2000	5195.53 draws/s	0:00:00	0:00:00
	2000	5966.73 draws/s	0:00:00	0:00:00
	2000	5088.98 draws/s	0:00:00	0:00:00

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 0 secc
There were 1043 divergences after tuning. Increase `target_accept` or reparameterize.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during samplir
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is r
```

No-U-turn sampler (NUTS)

This is a variation of Hamiltonian monte carlo that automatically tunes the number of leapfrog steps to allow more effective exploration of the parameter space.

Specifically, it uses a tree based algorithm that tracks trajectories forwards and backwards in time. The tree expands until a maximum depth is achieved or a “U-turn” is detected.



NUTS also does not use a metropolis step to select the final parameter value, instead the sample is chosen among the valid candidates along the trajectory.

NUTS

Model | Summary | Traces | Trajectories | ACF

```
with banana:
  nuts = pm.sample(
    draws=1000, tune=1000,
    step=pm.NUTS([x1,x2]),
    random_seed=1234
  )
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [x1, x2]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elap
	2000	64	0.09	11	4550.01 draws/s	0:00
	2000	41	0.25	3	3699.27 draws/s	0:00
	2000	10	0.16	51	3327.32 draws/s	0:00
	2000	10	0.17	11	3804.41 draws/s	0:00

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 1 secc
There were 125 divergences after tuning. Increase `target_accept` or reparameterize.
The rhat statistic is larger than 1.01 for some parameters. This indicates problems during samplir
The effective sample size per chain is smaller than 100 for some parameters.  A higher number is r
```

Some considerations

- Hamiltonian MC methods are all very sensitive to the choice of their tuning parameters (NUTS less so, but adds additional parameters)
- Hamiltonian MC methods require the gradient of the log density of the parameter of interest for the leapfrog integrator - limits this method to continuous parameters
- HMC updates are generally more expensive computationally than MH updates, but they also tend to produce chains with lower autocorrelation. Best to think about performance in terms of effective samples per unit of time.

Divergent transitions

Using Stan or PyMC with NUTS you will often see messages/ warnings about divergent transitions or divergences.

This is based on the assumption of conservation of energy with regard to the Hamiltonian system - this tells us that $H(\theta, \rho)$ should remain constant for the "particle" along its trajectory. When $H(\theta, \rho)$ of the trajectory diverges from its initial value then a divergence is considered to have occurred and positions after that point cannot be considered as the next draw.

The proximate cause of this is a break down of the first order approximations in the leapfrog algorithm.

The ultimate cause is usually a highly curved posterior or a posterior where the rate of curvature is changing rapidly.

Solutions?

Very much depend on the nature of the problem - typically we can potentially reparameterize the model and or adjust some of the tuning parameters to help the sampler deal with the problematic posterior.

For the latter the following options can be passed to `pm.sample()` or `pm.NUTS()`:

- `target_accept` - step size is adjusted to achieve the desired acceptance rate (larger values result in smaller steps which often work better for problematic posteriors)
- `max_treedepth` - maximum depth of the trajectory tree
- `step_scale` - the initial guess for the step size (scaled down by based on the dimensionality of the parameter space)

NUTS (adjusted)

Model | Summary | Traces | Trajectories | ACF

```
1 with banana:
2   nuts2 = pm.sample(
3     draws=1000, tune=1000,
4     step=pm.NUTS([x1,x2], target_accept=0.9),
5     random_seed=1234
6   )
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [x1, x2]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling S
	2000	0	0.08	63	2473.91 dr
	2000	0	0.12	87	2420.99 dr
	2000	1	0.06	7	2411.84 dr
	2000	0	0.08	7	2463.83 dr

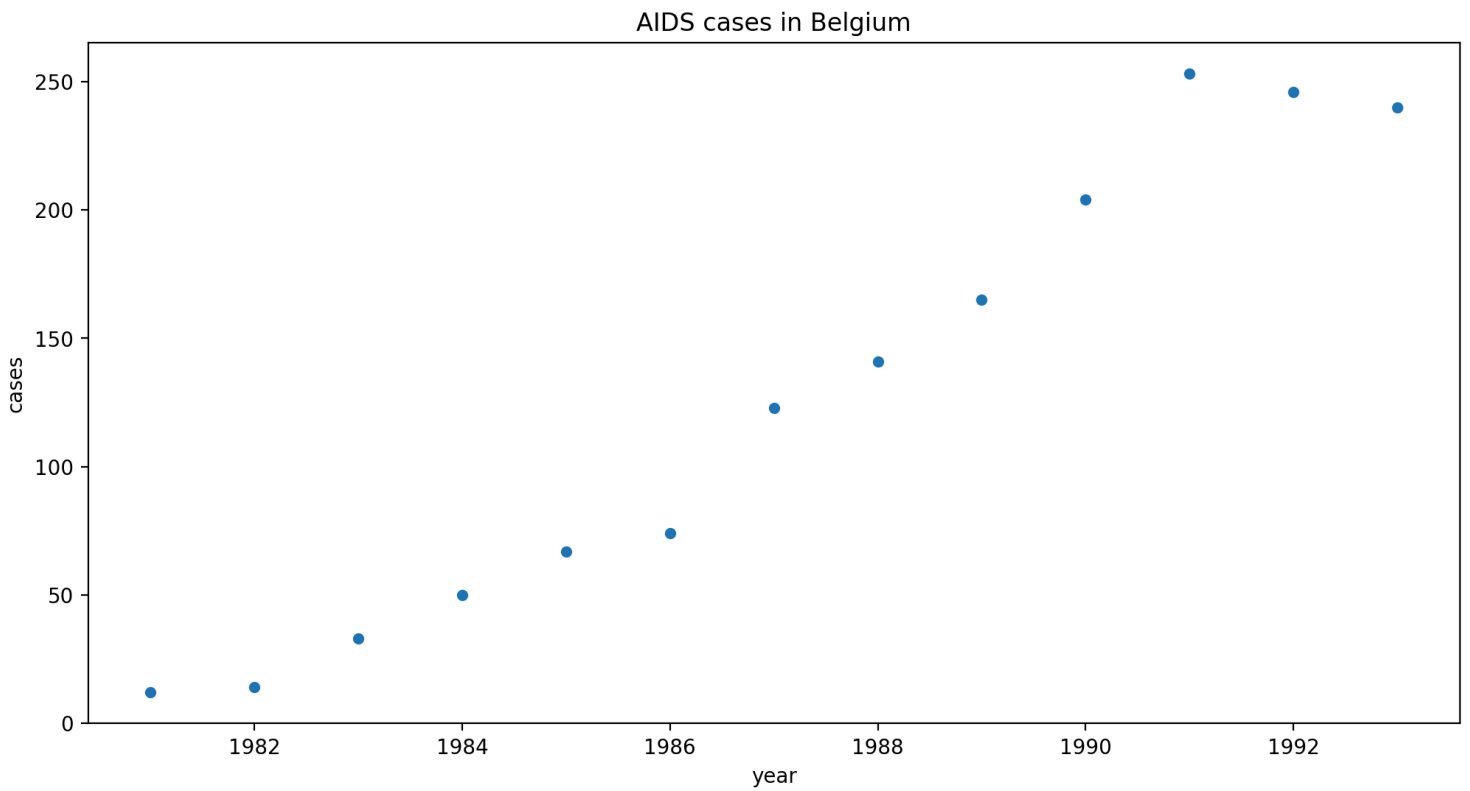
```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total)
There were 1 divergences after tuning. Increase `target_accept` or reparameterize.
The effective sample size per chain is smaller than 100 for some parameters.  A higher
```

Example 1 - Poisson Regression

Data

```
1 aids
```

	year	cases
0	1981	12
1	1982	14
2	1983	33
3	1984	50
4	1985	67
5	1986	74
6	1987	123
7	1988	141
8	1989	165
9	1990	204
10	1991	253
11	1992	246
12	1993	240



Model

```
y, X = patsy.dmatrices("cases ~ year", aids)

X_lab = X.design_info.column_names
y = np.asarray(y).flatten()
X = np.asarray(X)

with pm.Model(coords = {"coeffs": X_lab}) as model:
    b = pm.Cauchy("b", alpha=0, beta=1, dims="coeffs")
    η = X @ b
    λ = pm.Deterministic("λ", np.exp(η))

    likelihood = pm.Poisson("y", mu=λ, observed=y)

    post = pm.sample(random_seed=1234)
```

```
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [b]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling S
———	2000	0	0.00	7	7109.87 dr
———	2000	0	0.00	1	5394.50 dr
———	2000	0	0.00	1023	144.44 dra

```
2000    0              0.00          1023          170.17 dra
```

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total)
Chain 2 reached the maximum tree depth. Increase `max_treedepth`, increase `target_acc
Chain 3 reached the maximum tree depth. Increase `max_treedepth`, increase `target_acc
The rhat statistic is larger than 1.01 for some parameters. This indicates problems du
The effective sample size per chain is smaller than 100 for some parameters.  A higher
```

Summary

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_b
b[Intercept]	-9.810600e+01	1.696410e+02	-404.199	1.750000e-01	8.438100e+01	48.793	4.0
b[year]	1.220000e-01	7.600000e-02	0.002	2.060000e-01	3.800000e-02	0.018	5.0
λ[0]	2.644633e+146	4.581211e+146	23.261	1.057853e+147	2.281425e+146	NaN	4.0
λ[1]	3.137637e+146	5.435226e+146	28.930	1.255055e+147	2.706721e+146	NaN	4.0
λ[2]	3.722546e+146	6.448444e+146	35.980	1.489018e+147	3.211299e+146	NaN	4.0
λ[3]	4.416491e+146	7.650543e+146	44.749	1.766596e+147	3.809940e+146	NaN	4.0
λ[4]	5.239800e+146	9.076734e+146	55.654	2.095920e+147	4.520177e+146	NaN	4.0
λ[5]	6.216587e+146	1.076879e+147	69.218	2.486635e+147	5.362814e+146	NaN	4.0
λ[6]	7.375464e+146	1.277628e+147	86.015	2.950186e+147	6.362534e+146	NaN	4.0
λ[7]	8.750375e+146	1.515799e+147	106.700	3.500150e+147	7.548618e+146	NaN	4.0
λ[8]	1.038159e+147	1.798369e+147	131.889	4.152637e+147	8.955808e+146	NaN	4.0
λ[9]	1.231690e+147	2.133616e+147	135.458	4.926759e+147	1.062532e+147	NaN	4.0
λ[10]	1.461298e+147	2.531358e+147	135.780	5.845190e+147	1.260606e+147	NaN	4.0
λ[11]	1.733708e+147	3.003246e+147	136.104	6.934832e+147	1.495604e+147	NaN	4.0
λ[12]	2.056901e+147	3.563102e+147	136.428	8.227603e+147	1.774410e+147	NaN	4.0

Sampler stats

```
1 print(post.sample_stats)
```

```
<xarray.Dataset> Size: 496kB
Dimensions:                (chain: 4, draw: 1000)
Coordinates:
  * chain                  (chain) int64 32B 0 1 2 3
  * draw                   (draw) int64 8kB 0 1 2 3 4 5 ... 995 996 997 998 999
Data variables: (12/17)
    acceptance_rate        (chain, draw) float64 32kB 1.0 1.0 ... 0.9937 0.9483
    diverging              (chain, draw) bool 4kB False False ... False False
    energy                 (chain, draw) float64 32kB 4.669e+148 ... 96.27
    energy_error           (chain, draw) float64 32kB 0.0 0.0 ... 0.02436
    index_in_trajectory    (chain, draw) int64 32kB -1 -1 1 -3 ... 2 4 1005 254
    largest_eigval         (chain, draw) float64 32kB nan nan nan ... nan nan
    ...                     ...
    process_time_diff      (chain, draw) float64 32kB 3e-05 2.9e-05 ... 0.01391
    reached_max_treedepth  (chain, draw) bool 4kB False False ... True True
    smallest_eigval        (chain, draw) float64 32kB nan nan nan ... nan nan
    step_size              (chain, draw) float64 32kB 8.018e-77 ... 0.001951
    step_size_bar          (chain, draw) float64 32kB 1.257e-93 ... 0.001808
    tree_depth             (chain, draw) int64 32kB 1 1 1 3 1 1 ... 2 4 4 10 10
Attributes:
```

Tree depth

```
1 post.sample_stats["tree_depth"].values
```

```
array([[ 1,  1,  1,  3,  1,  1,  4,  2,  1,  1,  4,  1,  1,  1,  5,  1,  1,  1,  1,  3
         1,  4,  3,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  3],
       [ 1,  5,  1,  1,  5,  5,  1,  5,  1,  4,  1,  1,  1,  1,  1,  1,  1,  2,  1,  1
         4,  1,  1,  2,  2,  2,  1,  2,  1,  4,  1,  1,  2,  1],
       [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10
        10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10,  2,  5, 10, 10, 10, 10, 10,  2, 10,  9,  2, 10, 10
         2,  9, 10,  3, 10,  8, 10,  2,  7,  2,  4,  4, 10, 10]], shape=(4, 1000))
```

```
1 post.sample_stats["reached_max_treedepth"].values
```

```
array([[False, False, False, False, False, False, False, False, False, False, False, F
        False, False, False, ..., False, False, False, False, False, False, False, Fal
        False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False, False, False, F
        False, False, False, ..., False, False, False, False, False, False, False, Fal
        False, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True, ...,  True,  True,  True,  True,  True,  True,  True,  Tr
         True,  True,  True,  True,  True,  True,  True],
       [False,  True,  True,  True,  True,  True,  True, False, False,  True,  True, F
        False, False,  True, ...,  True, False, False, False, False, False,  True,  Tr
        False, False, False, False, False,  True,  True]], shape=(4, 1000))
```

Adjusting the sampler

```
with model:
  post = pm.sample(
    random_seed=1234,
    step = pm.NUTS(max_treedepth=20)
  )
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [b]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling
————————————	2000	0	0.00	91	102.77 dr
————————————	2000	0	0.00	1	112.30 dr
————————————	2000	0	0.00	3	106.68 dr
————————————	2000	0	0.00	2071	102.79 dr

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total)
The rhat statistic is larger than 1.01 for some parameters. This indicates problems du
```

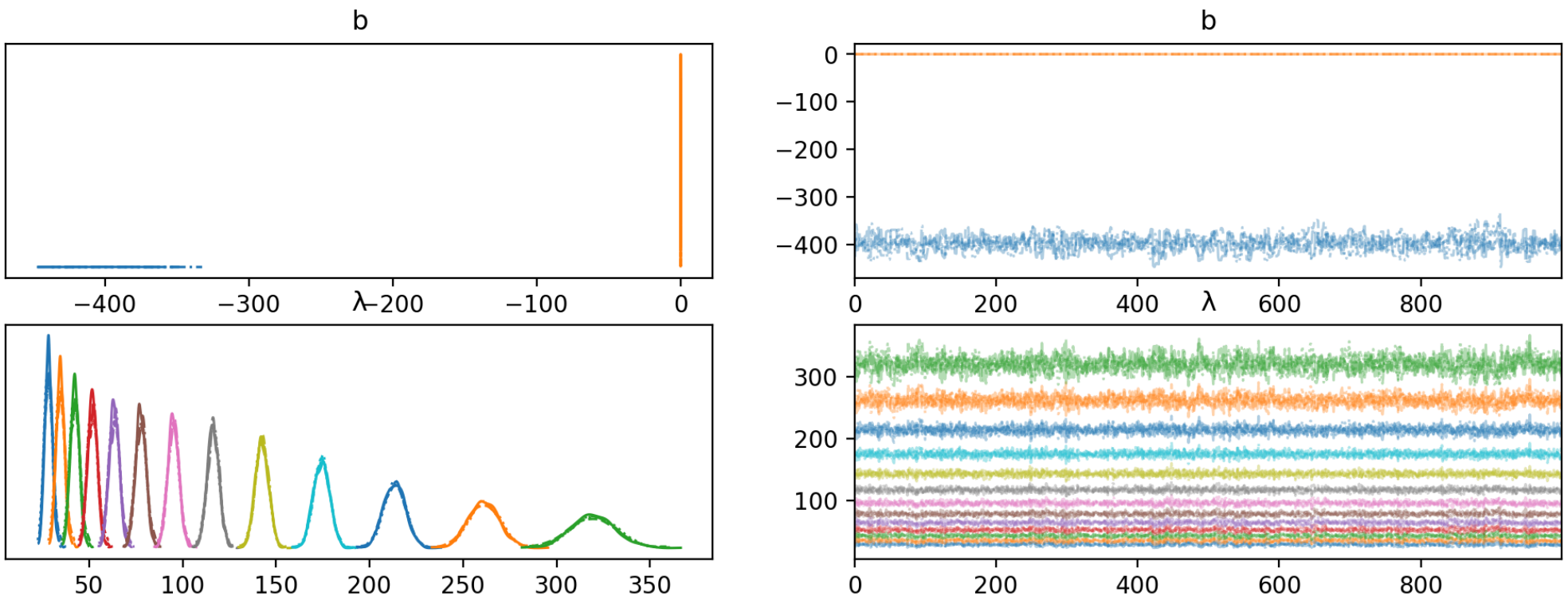
Summary

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b[Intercept]	-397.014	16.482	-430.085	-368.857	0.673	0.522	607.0	714.0	1.01
b[year]	0.202	0.008	0.188	0.219	0.000	0.000	607.0	714.0	1.01
λ[0]	28.354	2.120	24.271	32.130	0.083	0.064	676.0	839.0	1.01
λ[1]	34.685	2.322	30.344	38.973	0.089	0.067	693.0	886.0	1.01
λ[2]	42.433	2.516	37.650	47.018	0.094	0.070	719.0	935.0	1.01
λ[3]	51.915	2.691	46.682	56.655	0.098	0.071	758.0	986.0	1.00
λ[4]	63.520	2.838	57.726	68.380	0.099	0.069	825.0	1132.0	1.00
λ[5]	77.726	2.955	72.235	83.386	0.096	0.063	961.0	1248.0	1.00
λ[6]	95.114	3.056	89.294	100.866	0.086	0.056	1262.0	1574.0	1.00
λ[7]	116.401	3.203	110.884	123.106	0.072	0.051	1977.0	2219.0	1.00
λ[8]	142.461	3.547	135.449	148.765	0.063	0.054	3206.0	2894.0	1.00
λ[9]	174.367	4.340	166.502	182.479	0.077	0.065	3190.0	2869.0	1.00
λ[10]	213.435	5.871	202.877	224.795	0.132	0.088	1965.0	2545.0	1.00
λ[11]	261.273	8.389	245.983	277.402	0.236	0.140	1271.0	2056.0	1.00
λ[12]	319.856	12.146	297.797	343.323	0.387	0.247	989.0	1579.0	1.00

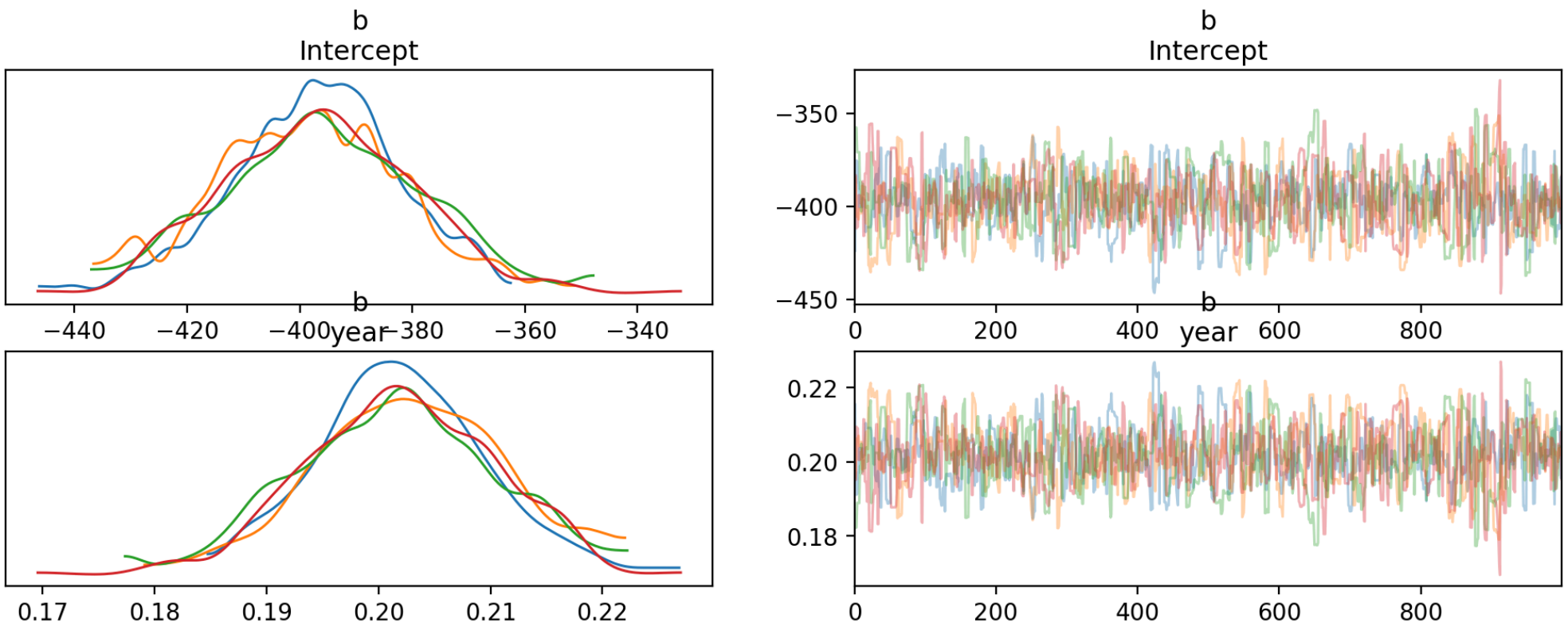
Trace plots

```
1 ax = az.plot_trace(post)
2 plt.show()
```



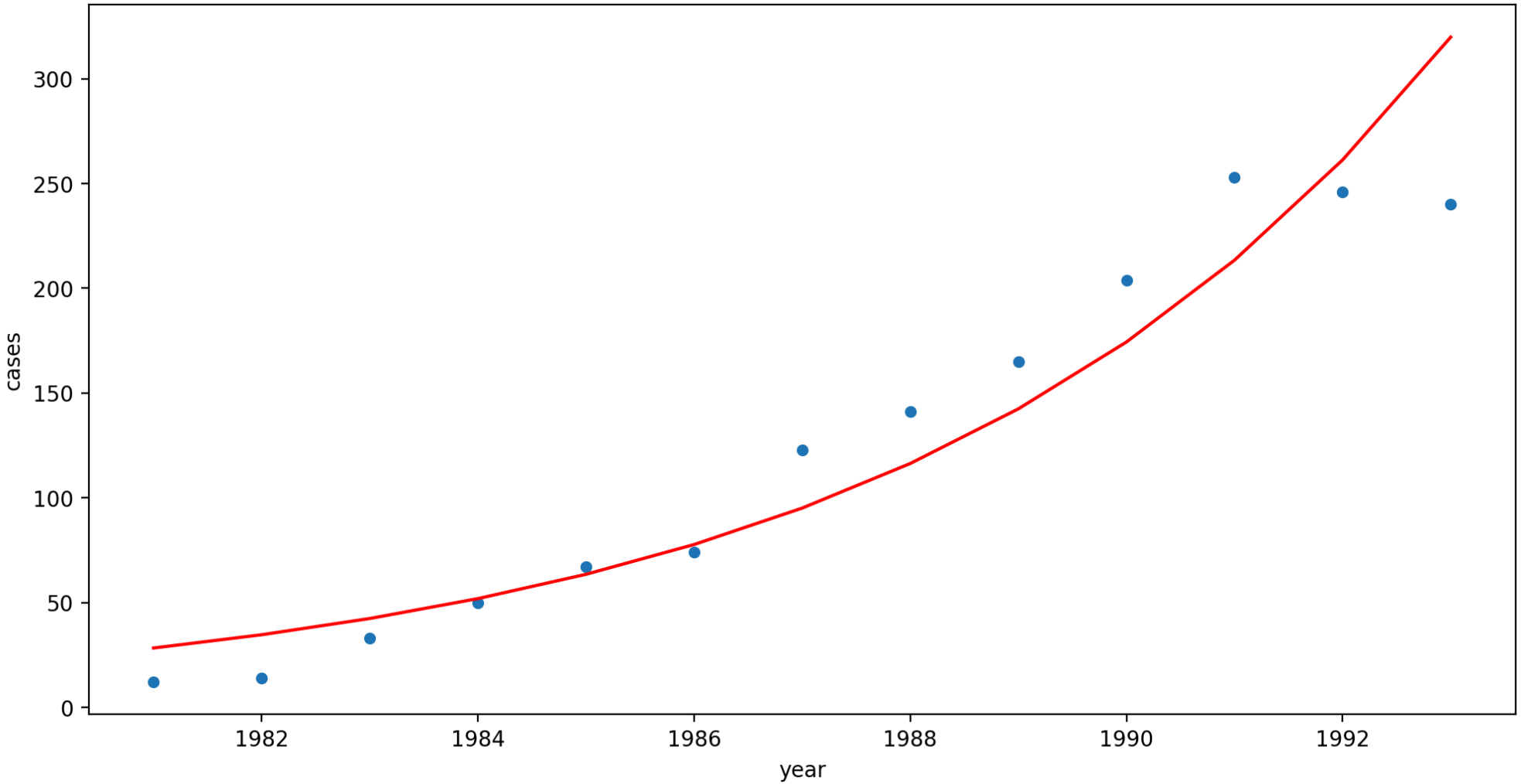
Trace plots (again)

```
1 ax = az.plot_trace(post.posterior["b"], compact=False)
2 plt.show()
```



Predictions (λ)

```
plt.figure(figsize=(12,6))
sns.scatterplot(x="year", y="cases", data=aids)
sns.lineplot(x="year", y=post.posterior["λ"].mean(dim=["chain", "draw"]), data=aids, color='r
plt.show()
```


300
250
200
150
100
50
0
cases
1982
1984
1986
1988
1990
1992
year

Revised model

```
y, X = patsy.dmatrices(
  "cases ~ year_min + I(year_min**2)",
  aids.assign(year_min = lambda x: x.year-np.min(x.year))
)

X_lab = X.design_info.column_names
y = np.asarray(y).flatten()
X = np.asarray(X)

with pm.Model(coords = {"coeffs": X_lab}) as model:
    b = pm.Cauchy("b", alpha=0, beta=1, dims="coeffs")
    η = X @ b
    λ = pm.Deterministic("λ", np.exp(η))

    likelihood = pm.Poisson("y", mu=λ, observed=y)

    post = pm.sample(random_seed=1234)
```

```
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [b]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elap
————————————	2000	0	0.08	47	2156.02 draws/s	0:00
————————————	2000	0	0.11	27	1679.65 draws/s	0:00
————————————	2000	0	0.06	23	2241.60 draws/s	0:00
————————————	2000	0	0.08	51	1855.25 draws/s	0:00

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 1 secc
```

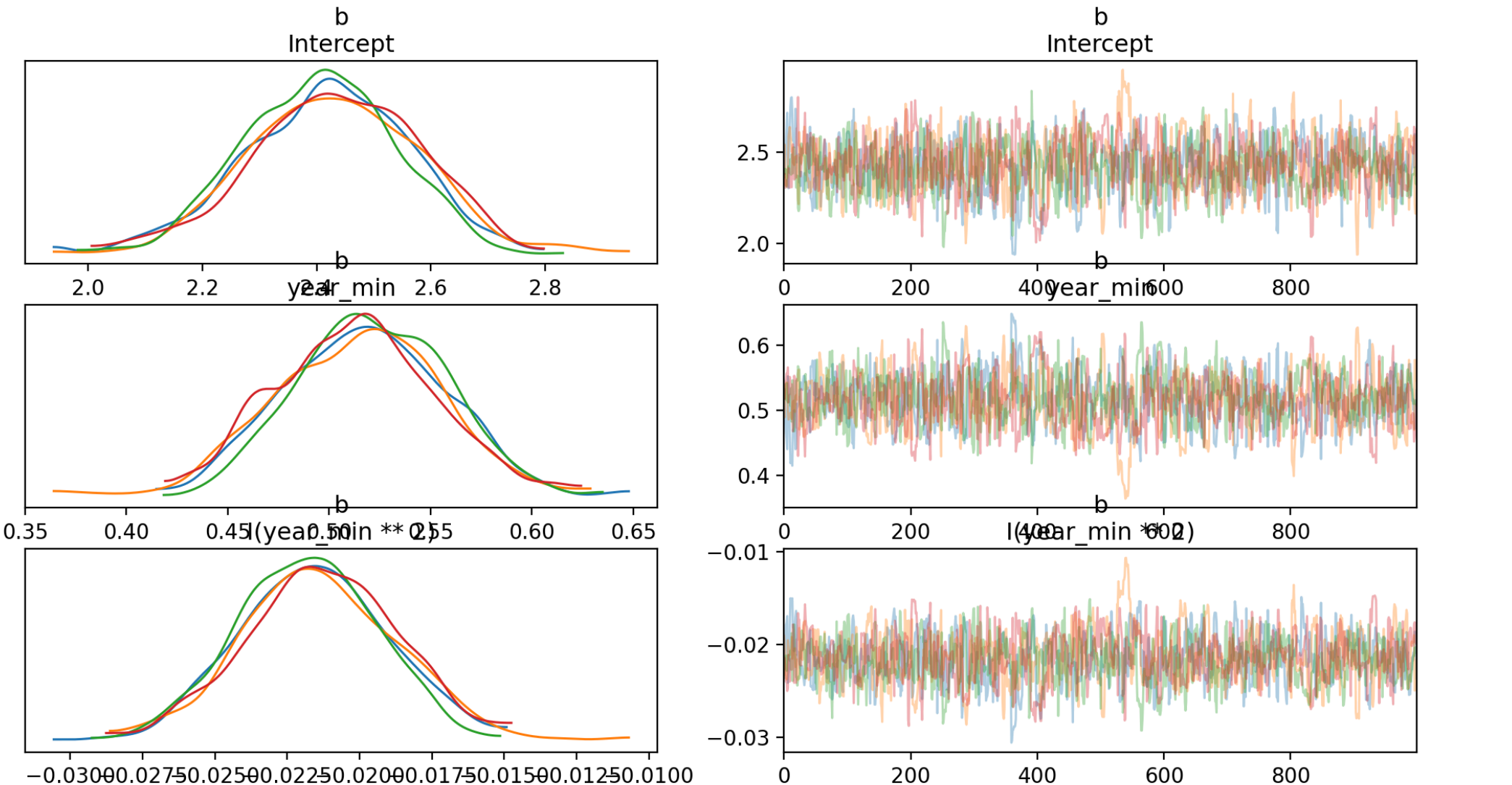
Summary

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b[Intercept]	2.421	0.143	2.157	2.682	0.005	0.004	747.0	772.0	1.01
b[year_min]	0.516	0.040	0.441	0.586	0.002	0.001	697.0	717.0	1.01
b[I(year_min ** 2)]	-0.022	0.003	-0.027	-0.017	0.000	0.000	728.0	748.0	1.01
λ[0]	11.377	1.628	8.494	14.397	0.060	0.054	747.0	772.0	1.01
λ[1]	18.583	2.023	14.951	22.386	0.072	0.059	806.0	814.0	1.01
λ[2]	29.123	2.362	24.722	33.503	0.077	0.060	940.0	937.0	1.00
λ[3]	43.770	2.644	38.808	48.609	0.073	0.059	1303.0	1229.0	1.00
λ[4]	63.061	2.976	57.519	68.538	0.066	0.062	2043.0	1376.0	1.00
λ[5]	87.064	3.553	80.530	93.737	0.070	0.066	2535.0	1858.0	1.00
λ[6]	115.161	4.438	106.590	123.213	0.095	0.072	2163.0	1938.0	1.00
λ[7]	145.916	5.408	136.611	156.803	0.133	0.086	1662.0	1765.0	1.00
λ[8]	177.090	6.089	166.647	189.286	0.159	0.102	1466.0	1435.0	1.00
λ[9]	205.864	6.270	194.661	218.133	0.154	0.103	1661.0	2028.0	1.00
λ[10]	229.247	6.449	217.633	242.082	0.118	0.099	3006.0	3109.0	1.00
λ[11]	244.590	8.201	229.959	261.102	0.148	0.122	3081.0	2803.0	1.00
λ[12]	250.092	12.377	225.629	272.498	0.293	0.217	1804.0	2326.0	1.00

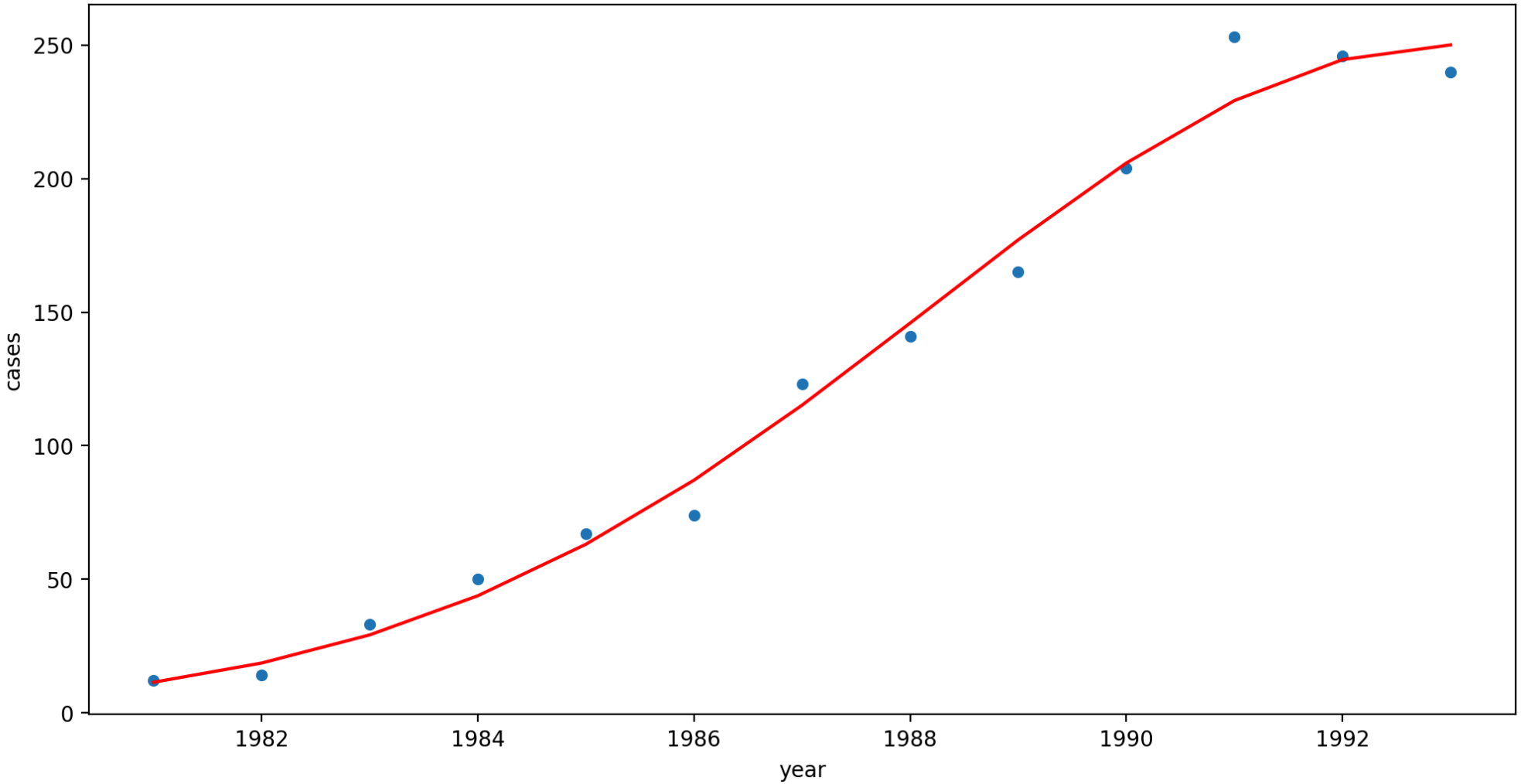
Trace plots

```
1  ax = az.plot_trace(post.posterior["b"], compact=False)
2  plt.show()
```



Predictions (λ)

```
plt.figure(figsize=(12,6))
sns.scatterplot(x="year", y="cases", data=aids)
sns.lineplot(x="year", y=post.posterior["λ"].mean(dim=["chain", "draw"]), data=aids, color='r
plt.show()
```


250
200
150
100
50
0
cases
1982
1984
1986
1988
1990
1992
year

Example 2 - Compound Samplers

Model with a discrete parameter

```
 1 import pytensor
 2
 3 n = pytensor.shared(np.asarray([10, 20]))
 4 with pm.Model() as m:
 5     p = pm.Beta("p", 1.0, 1.0)
 6     i = pm.Bernoulli("i", 0.5)
 7     k = pm.Binomial("k", p=p, n=n[i], observed=4)
 8
 9     step = pm.CompoundStep([
10       pm.NUTS([p]),
11       pm.BinaryMetropolis([i])
12     ])
13
14     trace = pm.sample(
15       1000, step=step
16     )
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>NUTS: [p]
>BinaryMetropolis: [i]
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elap
————————————	2000	0	1.07	1	8029.59 draws/s	0:00
————————————	2000	0	1.34	3	8120.01 draws/s	0:00
————————————	2000	0	0.93	3	7791.41 draws/s	0:00
————————————	2000	0	1.14	3	7584.69 draws/s	0:00

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws total) took 0 secc
```

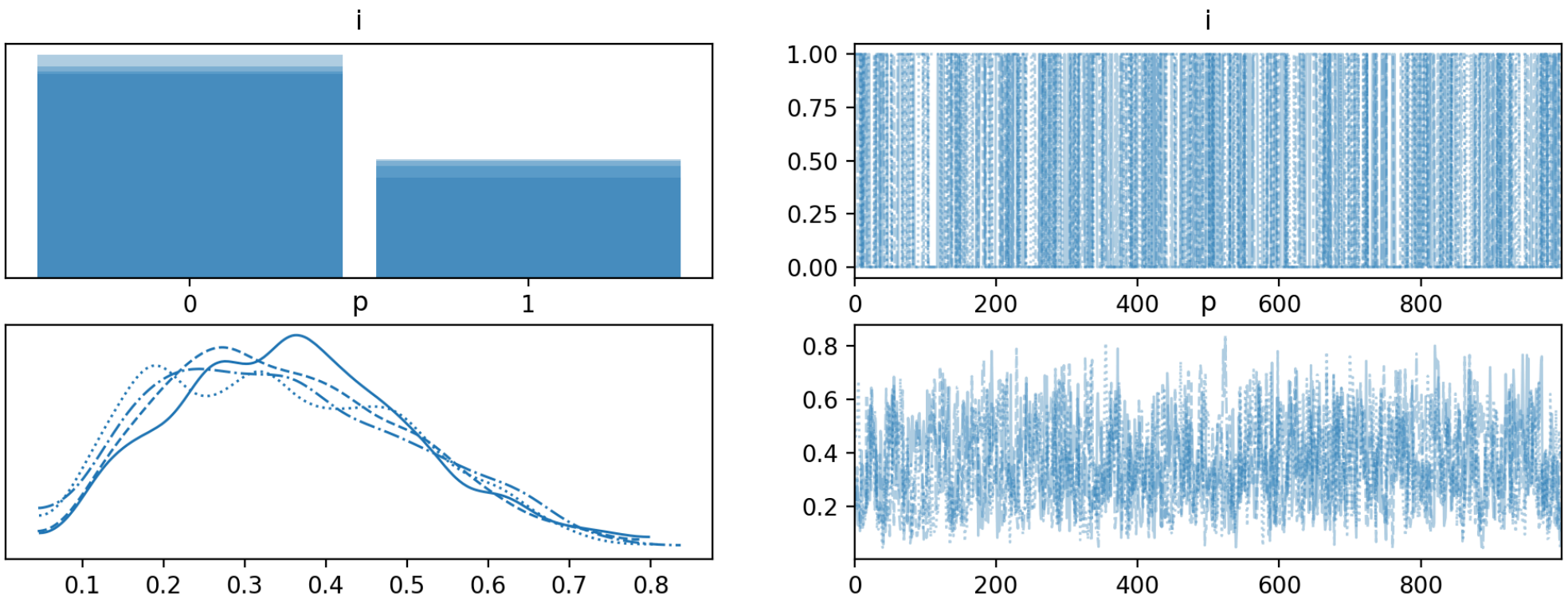
Summary

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
i	0.346	0.476	0.000	1.000	0.017	0.005	807.0	807.0	1.01
p	0.352	0.151	0.104	0.634	0.005	0.002	785.0	1910.0	1.01

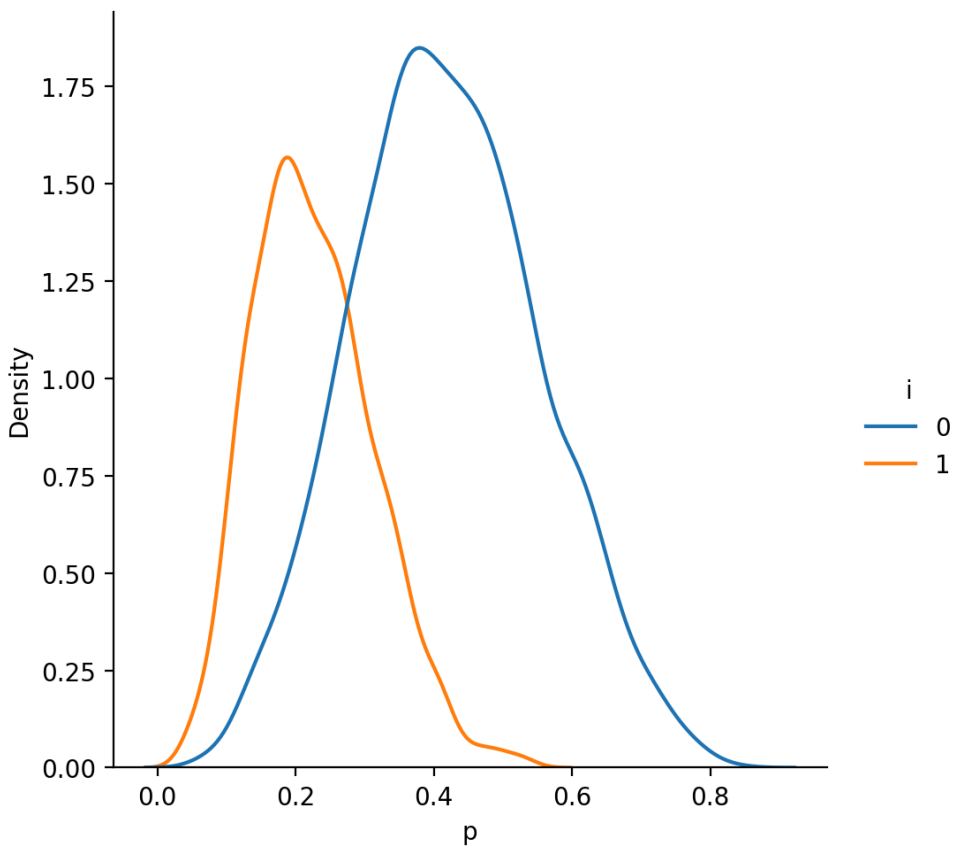
Trace plots

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



```python
d = pd.DataFrame({
  "p": trace.posterior["p"].values.flatten(),
  "i": trace.posterior["i"].values.flatten()
})
```

```python
sns.displot(d, x="p", hue="i", kind="kde")
plt.show()
```




```python
d.groupby("i").mean()
```

i	p
0	0.419230
1	0.226157

If we assume `i=0`:

$$p|x = 4, i = 0 \sim \text{Beta}(5, 7)$$

$$E(p|x = 4, i = 0) = \frac{5}{5 + 7} = 0.416$$

If we assume `i=1`:

$$p|x = 4, i = 1 \sim \text{Beta}(5, 17)$$

$$E(p|x = 4, i = 0) = \frac{5}{5 + 17} = 0.227$$