

Bayesian Regression Analysis

A Complete, Beginner-Friendly Guide

From classical regression to priors, likelihoods, MCMC,
posterior predictive checks, and multiple regression

Table of Contents

1. Why Bayesian Regression?	3
2. Building a Bayesian Regression Model	4
3. MCMC Diagnostics: Can You Trust the Posterior?	6
4. Making Predictions	8
5. Posterior Predictive Checks (PPC)	9
6. Multiple Linear Regression	10
7. Robust Regression: Handling Outliers with a Student-t Likelihood	13
8. Hierarchical (Multilevel) Models	14
9. Deciding Whether to Add or Drop a Variable	15
10. Probability Distributions for Priors and Likelihoods	18
11. PyMC Parameter Reference	20

1. Why Bayesian Regression?

A classical (frequentist) regression estimates a single fixed number for each coefficient and calls it a day. Bayesian regression instead treats every unknown quantity as something we're uncertain about — before seeing data we describe that uncertainty with a probability distribution, and after seeing data we update it. This section builds the intuition using a simple advertising-and-sales example.

1.1 The example data

Advertising spend (in \$1,000s) versus monthly sales (in \$1,000s):

Advertising (\$1,000)	Sales (\$1,000)
1	8
2	10
3	13
4	16
5	18
6	21
7	22
8	26

1.2 Classical regression

The classical model is:

$$Sales = \beta_0 + \beta_1 \times Advertising + \varepsilon$$

where β_0 is the intercept, β_1 is the slope, and ε is random error. After fitting, classical regression reports a single number, e.g. "the slope is 2.31," and treats it as fixed but unknown.

1.3 Bayesian thinking

Bayesian regression instead says: we are uncertain about β_1 . Before seeing any data we represent that uncertainty with a prior distribution — for example, "every additional \$1,000 of advertising probably increases sales by somewhere between 0 and 5 thousand dollars." After observing data, that belief is updated into a posterior distribution. So instead of a single number like $\beta_1 = 2.31$, we get $\beta_1 \sim$ a distribution centered around 2.3 — which tells us both the most likely value and how confident we should be in it.

1.4 Classical vs. Bayesian regression

Classical	Bayesian
Parameters are fixed	Parameters are random variables
Uses data only	Uses prior + data
Confidence interval	Credible interval
One estimate	Entire distribution
Uses MLE / OLS	Uses Bayes' theorem

1.5 The core idea

Everything in Bayesian statistics comes from one equation:

$$\text{Posterior} \propto \text{Likelihood} \times \text{Prior}$$

More precisely:

$$P(\theta \mid \text{Data}) = P(\text{Data} \mid \theta) \times P(\theta) / P(\text{Data})$$

where θ represents the parameters, the prior is what we believed beforehand, the likelihood is how probable the data are under those parameter values, and the posterior is our updated belief. The denominator $P(\text{Data})$ simply ensures the posterior integrates to 1; modern Bayesian software usually avoids computing it directly. The simplest way to remember it:

$$\text{New belief} = \text{Old belief} + \text{Evidence}$$

2. Building a Bayesian Regression Model

We'll now build the model for the same advertising/sales data step by step: choosing the model form, assigning priors, defining the likelihood, and computing the posterior.

2.1 Step 1 — Choose the model

The regression equation is exactly the same as the classical one:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad \varepsilon_i \sim \text{Normal}(0, \sigma)$$

This means the errors follow a Normal distribution.

2.2 Step 2 — Choose priors

A common beginner choice is weakly informative priors — wide enough to let the data dominate, but not so wide that sampling misbehaves.

Parameter	Prior	Meaning
Intercept β_0	Normal(0, 10)	Most likely around zero, but values of ± 20 or more are still possible
Slope β_1	Normal(0, 5)	Unknown, but values near zero are somewhat more likely than extreme values
Noise σ	HalfNormal(5)	Must be positive — this is a standard deviation

2.3 Step 3 — Prior predictive check

Before letting real data anywhere near the model, it's worth pausing to ask one more question: do these priors even make sense? A prior predictive check answers this by simulating fake data directly from the priors alone — no observed sales involved yet — and looking at whether the simulated outcomes are remotely plausible.

Concretely: draw a β_0 , β_1 , and σ from their priors, plug them into the same regression equation, and generate a fake dataset. PyMC does this in one call, before any fitting happens:

```
with pm.Model() as model:
    beta0 = pm.Normal("Intercept", mu=0, sigma=10)
    beta1 = pm.Normal("Slope", mu=0, sigma=5)
    sigma = pm.HalfNormal("Sigma", sigma=5)

    mu = beta0 + beta1 * x
    sales = pm.Normal("Sales", mu=mu, sigma=sigma, observed=y)

    prior_checks = pm.sample_prior_predictive(samples=500)
```

If these priors imply that sales could plausibly be -400 or $+5,000$ at an advertising spend of $\$3,000$, that's a sign the priors are too wide — even though nothing here has looked at real data yet. If the simulated range instead looks more like "sales somewhere between roughly -20 and 60 ," that's a reasonable starting point for genuinely uncertain, weakly informative priors.

This step catches a common beginner mistake: priors that look modest written down on paper, like $\text{Normal}(0, 10)$, can imply wildly unrealistic outcomes once combined through the regression equation. Checking this before fitting saves you from later mistaking a bad prior for a data problem.

2.4 Step 4 — Likelihood

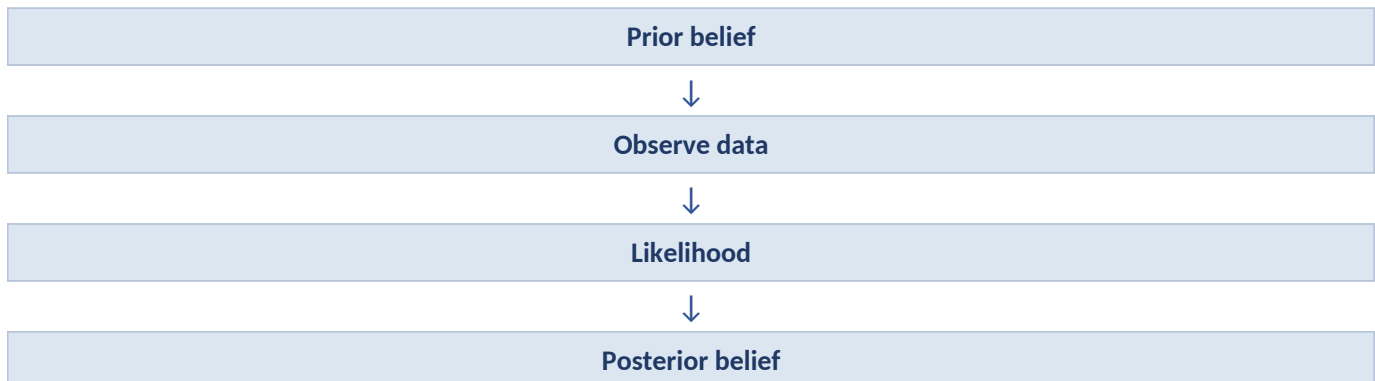
The likelihood describes how the observed data arise. For each observation:

$$y_i \sim \text{Normal}(\mu_i, \sigma), \quad \mu_i = \beta_0 + \beta_1 x_i$$

Every observed sales value is assumed to come from a Normal distribution centered on the regression line.

2.5 Step 5 — Posterior

Bayes combines the prior and the likelihood into the posterior:



For linear regression, the posterior usually has no simple closed-form solution once realistic priors are used. Modern Bayesian software instead uses MCMC (Markov Chain Monte Carlo) — most commonly the NUTS (No-U-Turn Sampler), an adaptive form of Hamiltonian Monte Carlo — to generate thousands of plausible parameter values that approximate the full posterior distribution. For example, thousands of slope samples such as 2.12, 2.28, 2.30, 2.41, 2.19, 2.33, ... together approximate the posterior for β_1 .

Visually, the prior starts out wide and uncertain; after seeing the data, the posterior narrows considerably around the values the data actually support. The data reduces uncertainty.

2.6 PyMC implementation

```
import pymc as pm
import numpy as np
import arviz as az

# Data
x = np.array([1, 2, 3, 4, 5, 6, 7, 8])
y = np.array([8, 10, 13, 16, 18, 21, 22, 26])

with pm.Model() as model:
    # Priors
    beta0 = pm.Normal("Intercept", mu=0, sigma=10)
    beta1 = pm.Normal("Slope", mu=0, sigma=5)
    sigma = pm.HalfNormal("Sigma", sigma=5)

    # Regression equation
    mu = beta0 + beta1 * x

    # Likelihood
    sales = pm.Normal("Sales", mu=mu, sigma=sigma, observed=y)

    # Posterior sampling
    trace = pm.sample(draws=2000, tune=1000, target_accept=0.9, random_seed=42)

az.summary(trace)
```

2.7 Posterior summary

A typical result (exact values vary slightly, since MCMC sampling is stochastic):

Parameter	Mean	SD	95% HDI
Intercept	5.8	1.0	[3.9, 7.7]
Slope	2.45	0.18	[2.11, 2.81]
Sigma	0.95	0.35	[0.45, 1.70]

2.8 How to read this

Intercept — mean 5.8, 95% HDI 3.9–7.7. Unlike a classical confidence interval, Bayesian statistics lets us say directly: given the model and priors, there is a 95% probability that the intercept lies between 3.9 and 7.7.

Slope — mean 2.45, credible interval 2.11–2.81. Every extra \$1,000 of advertising increases expected sales by about \$2,450, and we're fairly certain the true effect lies in this range under the model.

3. MCMC Diagnostics: Can You Trust the Posterior?

Running `pm.sample()` and getting numbers back doesn't automatically mean those numbers are trustworthy. MCMC is an approximation method, and like any approximation, it can fail quietly. This chapter extends the `r_hat` check already introduced in Chapter 2 into a fuller set of checks worth running before believing any posterior summary.

3.1 r_{hat} — do the chains agree?

As a quick recap: r_{hat} compares the variance between independent chains to the variance within each chain. A value at or very close to 1.00 (up to about 1.01) means the chains landed on the same distribution and agree with each other. Anything meaningfully higher — 1.05, 1.1 — means at least one chain explored a different part of the posterior, and nothing downstream from that parameter should be trusted yet.

3.2 Effective sample size (ESS)

MCMC samples are not independent draws — each one is correlated with the ones next to it in the chain, since the sampler moves step by step rather than jumping to a fresh random location each time. Effective sample size (ESS) estimates how many truly independent samples a correlated chain is actually worth.

`az.summary(trace)` reports two ESS columns: `ess_bulk` (how well the center of the distribution is estimated) and `ess_tail` (how well the extremes — the parts that define a credible interval — are estimated). As a rough rule of thumb, a combined ESS below roughly 400 suggests the posterior summary may be noisy and would benefit from more draws.

For example, if 8,000 posterior samples were requested but `ess_bulk` comes back at 150, only about 150 of those samples are doing the statistical work of independent draws — the rest are redundant, highly correlated repeats.

3.3 Divergences

A divergence is PyMC's way of flagging that the sampler tried to take a step and the underlying math broke down — usually because it hit a region of the posterior that curves very sharply (common in models with correlated or poorly-scaled parameters). A handful of divergences out of thousands of samples is sometimes tolerable, but any divergences at all are worth investigating rather than ignoring, since they can quietly bias the posterior.

```
trace = pm.sample(2000, tune=1000, target_accept=0.95, random_seed=42)
print(trace.sample_stats.diverging.sum().item(), "divergences")
```

If divergences show up, the two most common first fixes are raising `target_accept` (e.g. from 0.8 to 0.9 or 0.95, which makes the sampler take smaller, more cautious steps) or reconsidering the priors — very wide, weakly informative priors can sometimes make the geometry of the posterior harder to explore than tighter, more realistic ones.

3.4 Trace plots

Numbers alone can hide problems that are obvious visually. A trace plot shows, for each parameter, the sampled value at every iteration, side by side for every chain.

```
import arviz as az
az.plot_trace(trace, var_names=["Intercept", "Slope", "Sigma"])
```

A healthy trace plot looks like a flat, fuzzy band — often described as looking like "hairy caterpillars" — with all chains overlapping and no drift, trends, or one chain sitting apart from the others. A trace that drifts slowly upward, or where one chain sits at a different level than the rest, is a visual version of a poor r_{hat} and should be treated the same way: don't trust the summary yet.

3.5 A quick diagnostic checklist

Check	What it tells you	What to do if it fails
$r_{\text{hat}} \approx 1.00$	Chains agree with each other	Sample longer, or investigate the model

Check	What it tells you	What to do if it fails
ESS (bulk & tail) reasonably high	Samples aren't overly correlated	Increase draws, or reparameterize the model
Zero (or very few) divergences	The sampler explored the posterior geometry cleanly	Raise target_accept; reconsider priors or model structure
Trace plots look like flat, overlapping bands	No visible drift or disagreement between chains	Investigate whichever chain or parameter looks different

Only once all four look reasonable is it worth moving on to interpreting posterior means, credible intervals, or predictions — every earlier chapter implicitly assumed these checks would pass.

4. Making Predictions

Suppose Advertising = 10. We want:

$$\mu = \beta_0 + \beta_1 \times 10$$

But β_0 and β_1 each have thousands of posterior samples, not one value each. For every posterior draw we compute a separate prediction, for example:

Sample	Calculation	Prediction
1	$5.7 + 2.40 \times 10$	29.7
2	$6.1 + 2.52 \times 10$	31.3
3	$5.5 + 2.48 \times 10$	30.3

Repeating this across every posterior sample produces a posterior predictive distribution — a full range of plausible future sales values rather than a single point estimate.

4.1 Posterior prediction in PyMC

```
new_x = np.array([10])

posterior = trace.posterior
intercept = posterior["Intercept"].values.flatten()
slope = posterior["Slope"].values.flatten()

pred = intercept + slope * new_x[0]
print(pred.mean())
print(np.percentile(pred, [2.5, 97.5]))
```

A typical output:

- Mean prediction: 30.4
- 95% credible interval: [28.7, 32.2]

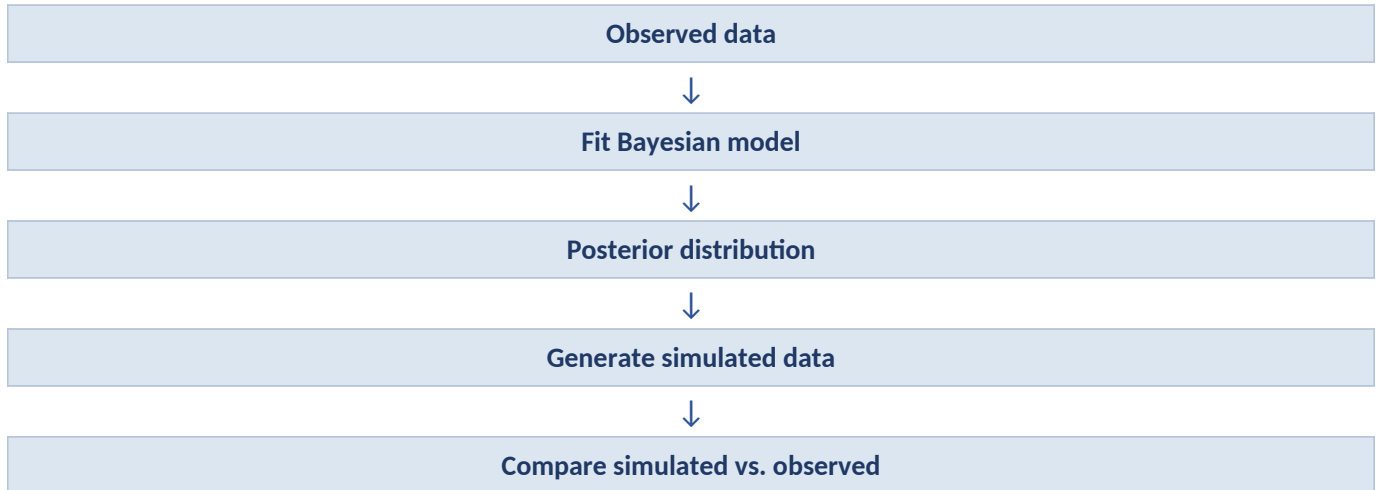
4.2 The complete Bayesian workflow



5. Posterior Predictive Checks (PPC)

After fitting a Bayesian model, converging isn't the same as being correct. Posterior Predictive Checks answer a more important question: does my model actually describe the data well? The intuition is simple — if the model is correct, it should be able to generate simulated data that looks like the data we actually observed.

5.1 The workflow



5.2 Worked example

Suppose one posterior sample gives Intercept = 5.5, Slope = 2.45, Sigma = 1.1. For Advertising = 4, the predicted mean is:

$$\mu = 5.5 + 2.45 \times 4 = 15.3$$

We then randomly generate plausible sales values around that mean (e.g. 15.7, 14.8, 16.1, 15.4). Repeating this for thousands of posterior samples produces many simulated datasets, which we compare against the real one:

	Values
Observed	8, 10, 13, 16, 18, 21, 22, 26
Simulation 1	8.3, 9.8, 13.5, 15.2, 17.6, 20.7, 23.1, 25.9
Simulation 2	7.6, 10.4, 12.9, 15.7, 18.9, 20.4, 22.5, 26.4

If the model is good, simulated datasets should resemble the real one across summary statistics (mean, standard deviation, min, max) and visual checks (histograms, scatter plots, regression lines). If observed values were 8, 10, 13, 16, 18, 21, 22, 26 but the model kept generating 4, 5, 7, 8, 9, 10, 11, 12, that would be a clear sign something is wrong — perhaps incorrect priors, a missing predictor, the wrong likelihood, or a nonlinear relationship.

5.3 In PyMC

```
with model:
    ppc = pm.sample_posterior_predictive(trace)

import arviz as az
az.plot_ppc(ppc)
```

The resulting plot overlays observed data against simulated data — good overlap means a well-fitting model. A useful analogy: if today's temperature was 31°C but a weather model kept predicting 5°C, you wouldn't trust it. Bayesian regression works the same way — a good model should be able to reproduce reality.

6. Multiple Linear Regression

So far we've used a single predictor (Advertising → Sales), but real business outcomes usually depend on several factors at once.

Variable	Meaning
Advertising	Marketing budget
Price	Product price
Discount	Discount offered
Season	Holiday or not
Competitor Price	Competitor's pricing

6.1 Classical model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \varepsilon$$

where x_1 = advertising, x_2 = price, x_3 = discount. For example, Sales = 20 + 2.1×Advertising – 1.8×Price + 0.9×Discount: every extra \$1,000 of advertising increases sales by 2.1 units, higher prices reduce sales, and more discount increases sales.

6.2 Bayesian version

The equation is identical, except every coefficient now gets a prior:

$$\beta_1, \beta_2, \beta_3 \sim \text{Normal}(0, 5), \quad \sigma \sim \text{HalfNormal}(5)$$

$$y_i \sim \text{Normal}(\mu_i, \sigma), \quad \mu_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{3i}$$

6.3 Example dataset

Advertising	Price	Discount	Sales
2	12	0	15
3	11	1	18
4	10	2	22
5	9	2	25
6	8	3	29
7	8	4	33

The model learns the advertising, price, and discount effects simultaneously. This matters because predictors often influence each other — for instance, if you advertise more during periods of higher discounts, the model estimates the effect of advertising while holding the other variables constant.

6.4 PyMC code

```
import pymc as pm
import numpy as np

advertising = np.array([2, 3, 4, 5, 6, 7])
price = np.array([12, 11, 10, 9, 8, 8])
discount = np.array([0, 1, 2, 2, 3, 4])
sales = np.array([15, 18, 22, 25, 29, 33])

with pm.Model() as model:
    beta0 = pm.Normal("Intercept", 0, 10)
    beta_adv = pm.Normal("Advertising", 0, 5)
    beta_price = pm.Normal("Price", 0, 5)
    beta_discount = pm.Normal("Discount", 0, 5)
    sigma = pm.HalfNormal("sigma", 5)

    mu = (beta0 + beta_adv * advertising
          + beta_price * price + beta_discount * discount)

    pm.Normal("Sales", mu=mu, sigma=sigma, observed=sales)
    trace = pm.sample(2000, tune=1000)
```

6.5 Example posterior summary

Parameter	Mean	95% HDI	Interpretation
Intercept	18.2	[14.7, 21.3]	Baseline sales when predictors are zero (often not directly meaningful if zero isn't realistic)
Advertising	2.05	[1.70, 2.42]	More advertising increases sales
Price	-1.62	[-2.08, -1.15]	Higher prices reduce sales
Discount	0.88	[0.51, 1.24]	Discounts increase sales

Notice that every coefficient has a distribution, not just a single value — this lets you quantify uncertainty directly for each predictor.

6.6 Why multiple regression is so powerful

Suppose two products both have high advertising, but Product A has high sales while Product B has low sales. Looking only at advertising, you might wrongly conclude advertising doesn't work. But if Product B also had a much higher price, adding Price to the model lets it separate the influence of advertising from the influence of pricing — isolating the effect of each predictor while accounting for the others.

6.7 Standardizing and centering predictors

Every prior in this guide so far has been written as something like $\text{Normal}(0, 5)$ without much thought about the scale of the predictor it applies to. That works fine when advertising spend ranges from 1 to 8, but it can quietly cause problems once predictors sit on very different scales — for instance, advertising in dollars (100 to 10,000) alongside a discount percentage (0 to 1).

A $\text{Normal}(0, 5)$ prior on the discount coefficient is fairly uninformative, allowing a wide range of plausible effect sizes. That same prior on the advertising coefficient, if advertising is left in raw dollars, effectively forces the coefficient toward implausibly tiny values — because with x in the thousands, even a modest-looking coefficient produces enormous swings in predicted sales.

The standard fix is to standardize continuous predictors before modeling — subtract the mean and divide by the standard deviation, so every predictor is centered at 0 with a spread of about 1:

$$x_{\text{standardized}} = (x - \text{mean}(x)) / \text{sd}(x)$$

```
advertising_std = (advertising - advertising.mean()) / advertising.std()  
price_std = (price - price.mean()) / price.std()
```

This has two benefits. First, the same weakly informative prior (Normal(0, 5), say) now means roughly the same thing for every predictor, regardless of its original units. Second, the intercept becomes directly interpretable: it's the expected sales when every predictor is at its average value, rather than when advertising and price are literally zero — a scenario that's often unrealistic or meaningless in a real business. Coefficients can always be converted back to the original units after fitting if needed, but most of the modeling and diagnostic work is easier to do on the standardized scale.

7. Robust Regression: Handling Outliers with a Student-t Likelihood

Every model in this guide so far has assumed Normal noise: $y_i \sim \text{Normal}(\mu_i, \sigma)$. That assumption has a specific weakness — the Normal distribution has very thin tails, so it treats any far-away point as extremely improbable. In practice, that makes ordinary Normal-likelihood regression quite sensitive to outliers: a single unusual data point can pull the estimated line noticeably toward it.

7.1 Why Normal likelihoods break down

Suppose one month had a data-entry error, or a genuine one-off event — advertising spend of \$4,000 but sales of only 2 (instead of the expected ~16). Under a Normal likelihood, the model has no good explanation for a point that far from the line except to shift the whole line and inflate σ to accommodate it. One bad data point can distort every coefficient.

7.2 The Student-t alternative

The Student-t distribution looks similar to the Normal — bell-shaped and centered on a mean — but has heavier tails, controlled by a parameter called the degrees of freedom (often written ν , "nu"). Heavier tails mean the model treats an occasional far-away point as merely unusual rather than nearly impossible, so it doesn't have to distort the whole line to accommodate it.

$$y_i \sim \text{StudentT}(\nu, \mu_i, \sigma), \quad \mu_i = \beta_0 + \beta_1 x_i$$

As ν grows large (roughly above 30), Student-t becomes nearly indistinguishable from Normal. Small values (around 3–7) give noticeably heavier tails and more robustness to outliers. Rather than picking ν by hand, it's common to let the model estimate it from the data with its own prior.

7.3 PyMC code

```
with pm.Model() as robust_model:
    beta0 = pm.Normal("Intercept", mu=0, sigma=10)
    beta1 = pm.Normal("Slope", mu=0, sigma=5)
    sigma = pm.HalfNormal("Sigma", sigma=5)
    nu = pm.Gamma("nu", alpha=2, beta=0.1) # degrees of freedom, > 0

    mu = beta0 + beta1 * x
    sales = pm.StudentT("Sales", nu=nu, mu=mu, sigma=sigma, observed=y)

    trace = pm.sample(2000, tune=1000, target_accept=0.95, random_seed=42)
```

Notice this changes only one line compared to the original model: `pm.Normal("Sales", ...)` becomes `pm.StudentT("Sales", nu=nu, ...)`. Everything else — the priors on the intercept and slope, the linear predictor `mu`, running `pm.sample()` — stays identical. This is a useful general pattern: swapping the likelihood is usually a small, localized change, not a full model rewrite.

7.4 Interpreting v

A small posterior mean for v (say, 4 or 5) is itself informative — it suggests the data genuinely has heavier tails than a Normal distribution would expect, which is evidence that outliers or unusual points are a real feature of this dataset, not just noise to explain away. A large posterior mean for v (say, above 30) suggests the Normal assumption was probably fine all along, and the extra flexibility of Student-t wasn't really needed.

8. Hierarchical (Multilevel) Models

Every model so far has pooled all the data into one shared line — one intercept, one slope, for everything. Real data often comes in groups: sales across several stores, ad performance across several regions, patients across several hospitals. Hierarchical models are the Bayesian answer to the question: should each group get its own line, or should they all share one?

8.1 The problem: to pool or not to pool

Suppose you have advertising and sales data from five different regions. There are three natural options.

Approach	What it assumes	Weakness
Complete pooling	Every region shares exactly the same intercept and slope — the model in every earlier chapter.	Ignores real differences between regions
No pooling	Fit a completely separate regression per region, with no information shared between them.	Regions with little data get noisy, unreliable estimates
Partial pooling (hierarchical)	Each region gets its own intercept and slope, but those are drawn from a shared distribution learned across all regions.	The middle ground — usually the best of both

8.2 How partial pooling works

Instead of a single slope, each region i gets its own slope $\beta_{1,i}$, but those region-level slopes are themselves assumed to come from a shared Normal distribution:

$$\beta_{1,i} \sim \text{Normal}(\mu_{\beta 1}, \tau_{\beta 1})$$

Here $\mu_{\beta 1}$ is the average slope across all regions, and $\tau_{\beta 1}$ controls how much regions are allowed to differ from that average. A region with abundant data mostly determines its own slope from its own data; a region with sparse or noisy data gets pulled ("shrunk") toward the overall average — borrowing strength from the other regions rather than being estimated in isolation. This shrinkage is the key benefit hierarchical models provide over fitting each group separately.

8.3 PyMC code

Suppose `region_idx` is an array of integers (0–4) indicating which region each observation belongs to, alongside the usual `spend` and `sales` arrays:

```
n_regions = 5
region_idx = np.array([...]) # 0-4, one entry per observation

with pm.Model() as hierarchical_model:
    # Population-level ("hyper") priors
    mu_beta1 = pm.Normal("mu_beta1", mu=0, sigma=5)
    tau_beta1 = pm.HalfNormal("tau_beta1", sigma=5)

    # Region-level intercepts and slopes
    beta0 = pm.Normal("beta0", mu=0, sigma=10, shape=n_regions)
    beta1 = pm.Normal("beta1", mu=mu_beta1, sigma=tau_beta1, shape=n_regions)
    sigma = pm.HalfNormal("sigma", sigma=5)

    mu = beta0[region_idx] + beta1[region_idx] * spend
    sales = pm.Normal("Sales", mu=mu, sigma=sigma, observed=sales_obs)

    trace = pm.sample(2000, tune=1000, target_accept=0.95, random_seed=42)
```

The `shape=n_regions` argument gives each region its own `beta0` and `beta1`, while `beta0[region_idx]` and `beta1[region_idx]` route each observation to its own region's coefficients during fitting. `mu_beta1` and `tau_beta1` are the hyperpriors — priors on the parameters that describe the population of regions, rather than on any single region directly.

8.4 Why this matters

Hierarchical models are often described as one of the biggest practical advantages of Bayesian methods, because they let every group benefit from the data of every other group without forcing them to be identical. A region with only two months of data doesn't get thrown out or trusted blindly — its estimate is automatically tempered by what's been learned from the other four regions, and that tempering happens naturally from the model structure rather than an ad hoc rule.

9. Deciding Whether to Add or Drop a Variable

In frequentist regression, the common rule is: is the p-value below 0.05? If not, remove the variable. Bayesian regression generally avoids single-threshold decisions like this, focusing instead on evidence, uncertainty, and predictive performance.

9.1 Step 1 — Does the posterior indicate a meaningful effect?

Variable	Mean	95% Credible Interval
TV	2.10	[1.60, 2.55]
Facebook	0.85	[0.30, 1.35]
Discount	0.05	[-0.45, 0.52]

Discount's interval contains zero, meaning the effect could reasonably be negative, zero, or positive — evidence of genuine uncertainty. This alone is not a reason to drop it automatically; ask the next questions first.

9.2 Step 2 — Is the effect practically important?

Even if an interval excludes zero (e.g. Discount mean 0.08, 95% CI [0.02, 0.14]), the size of the effect matters. If a 10% increase in discount changes sales by only 0.1%, the effect may be statistically real but too small to matter for decisions. This idea is often formalized with a Region of Practical Equivalence (ROPE), where effects close to zero are treated as practically negligible.

9.3 Step 3 — Does the variable improve prediction?

This is often the most important criterion. Compare a simpler model (TV + Price + Season) against a richer one (TV + Price + Season + Discount) using predictive-performance metrics:

- LOO (Leave-One-Out Cross-Validation) — often the preferred choice
- WAIC (Widely Applicable Information Criterion)
- Posterior predictive checks

If adding a variable barely changes predictive performance, it may not be worth the extra complexity.

9.4 Worked example: comparing models with `az.compare()`

This step is a short, mechanical one once the competing models are fit:

```
import arviz as az

with model_a: # TV + Price
    trace_a = pm.sample(2000, tune=1000, random_seed=42)
    pm.compute_log_likelihood(trace_a)

with model_b: # TV + Price + Discount
    trace_b = pm.sample(2000, tune=1000, random_seed=42)
    pm.compute_log_likelihood(trace_b)

comparison = az.compare({"Model A": trace_a, "Model B": trace_b})
print(comparison)
```

`az.compare()` returns a table ranked from best to worst expected predictive performance, along with `elpd_diff` (the estimated difference in predictive accuracy between each model and the best one) and its standard error, `dse`. The rule of thumb: if `elpd_diff` is smaller than roughly one or two times its own standard error, the models are practically indistinguishable in predictive terms — exactly the situation in the Model A vs. Model B example below, where a tiny LOO difference (900 vs. 899) wasn't worth the added complexity of keeping Discount in the model.

This is the same idea introduced in Step 3 above, just made concrete: `az.compare()` is the tool that actually produces the LOO/WAIC numbers this framework relies on, rather than something computed by hand.

9.5 Steps 5–7 — PPC, domain knowledge, and multicollinearity

- Posterior predictive checks: fit the model with and without the variable, and see which reproduces the observed data better.
- Domain knowledge: a variable with a weak statistical signal may still be kept if domain experts or prior research consistently support its effect.
- Multicollinearity: if two predictors are nearly identical (e.g. TV spending and total advertising), the model struggles to separate their effects — producing wide credible intervals and unstable estimates. In this case, one variable may be removed or the variables combined.

9.6 A practical example

Model	Variables (95% CI)	LOO score
A	TV [1.6, 2.4], Price [-1.8, -0.9]	900
B	TV [1.6, 2.4], Price [-1.8, -0.9], Discount [-0.3, 0.4]	899

Although Model B scores slightly better, the improvement is tiny and Discount's effect remains highly uncertain. Many analysts would choose Model A because it is simpler and predicts almost as well.

9.7 Bayesian vs. frequentist variable selection

Question	Frequentist	Bayesian
Is the coefficient different from zero?	p-value	Posterior distribution and credible interval
Should I keep the variable?	$p < 0.05$	Predictive performance + uncertainty + practical importance + domain knowledge
Model comparison	AIC / BIC	LOO, WAIC, Bayes factors (when appropriate)
Primary goal	Statistical significance	Prediction, uncertainty quantification, and scientific understanding

9.8 Recommended workflow

1. Start with theory — include variables expected to matter based on business knowledge.
2. Fit the Bayesian model.
3. Check MCMC diagnostics (R-hat, effective sample size, trace plots) to ensure the posterior is reliable.
4. Inspect posterior means and 95% credible intervals, considering practical effect sizes, not just whether the interval includes zero.
5. Run posterior predictive checks to verify the model reproduces observed data.
6. Compare competing models using LOO or WAIC — prefer the simpler model if removing a variable doesn't hurt predictive performance.
7. Choose the simplest model that predicts well and makes scientific or business sense.

10. Probability Distributions for Priors and Likelihoods

A common beginner question is "which prior should I use?" — the better question is "what type of variable am I modeling?" The distribution should match the nature of the data or parameter. The rule: first ask what your variable looks like, then choose a distribution that naturally describes that kind of variable.

10.1 The Bayesian modeling recipe

Every Bayesian model has two parts: unknown parameters get a prior distribution, and observed data get a likelihood distribution — the prior encodes belief before seeing data, and the likelihood encodes how the observed data are generated.

10.2 The six most important distributions

Distribution	Usually used for	Possible values
Normal	Continuous numbers	$(-\infty, +\infty)$
Bernoulli	Yes/No outcome	0 or 1
Poisson	Counts	0, 1, 2, ...
Beta	Probabilities	0–1
Gamma	Positive continuous quantities	> 0
Half-Normal	Positive parameters (especially standard deviations)	≥ 0

Normal distribution

The workhorse of Bayesian statistics — the familiar bell-shaped pattern (e.g. adult heights: most people near the average, very short/tall people rare). As a prior, it suits any real-valued coefficient, e.g. `pm.Normal("beta", mu=0, sigma=5)` — "probably near zero, but larger positive/negative values are possible." As a likelihood, it suits continuous measurements such as monthly sales.

Bernoulli distribution

Models binary outcomes with only two possibilities: clicked/didn't click, purchased/didn't purchase, win/lose. Each observation follows `Purchase ~ Bernoulli(p)`, where p is the underlying probability. Marketing example: predicting whether a customer clicks an ad.

Poisson distribution

Models counts during a fixed period — calls, purchases, accidents, website visits, complaints. Values are always non-negative integers (never 2.7 complaints). Example: `Complaints ~ Poisson( $\lambda$ )`, where λ is the average count. Marketing example: number of purchases, app installs, or store visits.

Beta distribution

Models numbers strictly between 0 and 1 — exactly what probabilities are. Used whenever the unknown parameter is a probability, such as a conversion rate: `p ~ Beta(2, 2)`. Different Beta priors encode different beliefs — `Beta(1,1)` is flat (every probability equally likely), while `Beta(20,5)` encodes a strong belief that the probability is near 0.8. Marketing example: estimating click-through rate (CTR).

Gamma distribution

Models positive continuous values such as waiting times, insurance claims, or rainfall — never negative. Gamma is also sometimes used as a prior for positive scale parameters, though Half-Normal and Half-Student-t are now more common choices for that role.

Half-Normal distribution

Simply the positive half of a Normal distribution — used whenever a quantity, like a standard deviation, can never be negative ($\sigma = -3$ is impossible). In regression, $\sigma \sim \text{HalfNormal}(5)$ is probably the most common prior beginners use for the residual standard deviation.

10.3 Putting it together

Response variable	Likelihood	Typical priors
Sales = Advertising + Price (continuous)	Normal	Normal for coefficients, Half-Normal for σ
Purchased? Yes/No	Bernoulli	Normal for coefficients
Number of purchases (0, 2, 5, 7, ...)	Poisson	Normal for coefficients

10.4 A simple decision tree

Whenever you build a Bayesian model, work through this sequence of questions about your response variable:

Is your variable...	Use...
Any real number (positive or negative)	Normal
Only 0 or 1	Bernoulli
A count (0, 1, 2, ...)	Poisson
A probability (between 0 and 1)	Beta
A positive continuous value	Gamma
A positive scale parameter (like σ)	Half-Normal

10.5 A marketing and business cheat sheet

Business problem	Target variable	Likelihood	Typical priors
Predict sales revenue	Continuous (e.g. \$125,000)	Normal	Normal for coefficients, Half-Normal for σ
Will a customer purchase?	Yes/No	Bernoulli	Normal for coefficients
How many purchases today?	Count (0, 1, 2, ...)	Poisson	Normal for coefficients
Estimate click-through rate	Probability (0–1)	Often Bernoulli with a Beta prior on p	Beta
Average delivery time	Positive continuous	Gamma (or Log-Normal)	Gamma
Residual standard deviation	Positive continuous parameter	—	Half-Normal

10.6 One final insight

The likelihood is chosen based on your response (target) variable, not your predictors. For example, when predicting monthly sales: the response (Sales) is continuous, so it gets a Normal likelihood; the predictors (Advertising, Price, Discount) can be continuous or categorical after encoding; the regression coefficients are unknown real-valued parameters, so they get Normal priors; and the residual standard deviation σ must be positive, so it gets a Half-Normal prior. Once you identify the type of your response variable and the constraints on your parameters, choosing appropriate likelihoods and priors becomes systematic rather than something to memorize.

11. PyMC Parameter Reference

Every code example in this guide reuses the same handful of PyMC building blocks. This section collects them in one place and explains what each parameter controls, so the code stops looking like a black box.

11.1 pm.Model()

```
with pm.Model() as model:
    ...
    trace = pm.sample(...)
```

This is a context manager, not a function with parameters to tune. Everything defined inside the with block — priors, the linear predictor, and the likelihood — is automatically registered onto model. You never have to manually wire pieces together; PyMC builds the full probability graph just from what's declared inside the block.

11.2 pm.Normal(name, mu, sigma, observed)

Used throughout this guide for priors on coefficients and for the likelihood. It takes the same four arguments in both roles — only whether observed is supplied changes what it means.

Parameter	Meaning	Typical use
name (string)	The label PyMC uses internally to identify this variable — shown in <code>az.summary()</code> and trace outputs. Must be unique within the model.	"Intercept", "Slope", "Sigma", "Sales"
mu	The mean of the Normal distribution.	0 for priors on coefficients (no reason to favor positive over negative beforehand); the linear predictor (e.g. $\text{beta0} + \text{beta1} * x$) for the likelihood
sigma	The standard deviation — how wide/uncertain the distribution is.	5–10 for weakly informative priors; left as an unknown parameter (its own <code>pm.HalfNormal</code>) for the likelihood's noise term
observed	Attaches real data to this variable, turning it from a prior into a likelihood term. Omit this argument entirely for priors.	<code>observed=y</code> or <code>observed=sales</code>

For example, `pm.Normal("Slope", mu=0, sigma=5)` declares a prior belief about the slope centered at zero with a fairly wide spread, while `pm.Normal("Sales", mu=mu, sigma=sigma, observed=y)` declares the likelihood: "the observed sales values are assumed to be draws from this Normal distribution."

11.3 pm.HalfNormal(name, sigma)

Used for the residual noise term σ , since a standard deviation can never be negative. It takes the same name argument as pm.Normal, plus one shape parameter:

Parameter	Meaning	Typical use
name (string)	Same role as in pm.Normal — the internal label.	"Sigma", "sigma"
sigma	Controls how large the noise is allowed to be. This is not the noise itself — it's the spread of the prior placed over the noise.	5 is a common weakly informative choice, wide enough to let the data pin down the actual noise level

11.4 pm.sample(draws, tune, chains, cores, target_accept, random_seed)

This runs MCMC (the NUTS sampler) and returns the posterior as a trace object. None of its arguments change the model itself — they control how thoroughly and reliably the posterior gets explored.

Parameter	Meaning	Typical value
draws	How many posterior samples to keep, per chain, after tuning finishes. More draws give smoother, more precise posterior summaries.	2000
tune	How many exploratory steps the sampler takes first to calibrate its step size, before the samples that get kept begin. These are discarded, not part of the posterior.	1000
chains	How many independent random walks to run. Comparing chains is exactly what r_hat checks — if they agree, you can trust the result.	4
cores	How many chains to run in parallel processes at once. Purely a speed setting with no effect on the statistical result.	1 in constrained environments; often left at the number of CPU cores available
target_accept	How cautious the sampler is about step size, expressed as a target acceptance rate. Raising it (e.g. to 0.9 or 0.95) takes smaller, safer steps — slower, but helps when the sampler reports divergences.	0.8 (PyMC default) to 0.95
random_seed	Fixes the random number generator so results are reproducible run to run.	42 (or any fixed integer)

So `pm.sample(draws=2000, tune=1000, chains=4, cores=1, random_seed=42)` runs 4 independent chains, each discarding its first 1,000 calibration steps and keeping 2,000 real samples — 8,000 posterior samples in total, run one chain at a time, with results that will be identical every time the code is re-run.

11.5 Reading the trace

`pm.sample()` returns a trace object holding all the posterior samples, organized by chain and draw. Two patterns recur throughout this guide:

Code	What it does
<code>az.summary(trace, var_names=["b0", "b1", "sigma"])</code>	Collapses the samples into a table: posterior mean, standard deviation, credible interval, and convergence diagnostics like r_hat, for each named parameter.

Code	What it does
<code>trace.posterior["Slope"].values.flatten()</code>	Pulls out every individual posterior sample for one parameter as a flat NumPy array — the raw material for computing predictions, probabilities like $P(\text{slope} > 0)$, or custom percentiles by hand.

11.6 pm.sample_posterior_predictive(trace)

Used for posterior predictive checks (Section 5). Rather than manually recomputing predictions from posterior samples, this function asks the model itself to simulate a full new dataset for every posterior draw — the same size and shape as the original observed data — which can then be compared against the real observations with `az.plot_ppc()`.

11.7 Putting it together

A useful way to read any PyMC model in this guide: the `pm.Model()` block declares what's unknown (priors) and how data is generated from those unknowns (the likelihood, marked by `observed=`); `pm.sample()`'s arguments only control the mechanics of exploring that model reliably; and everything downstream — summaries, predictions, probability statements, predictive checks — is just arithmetic performed on the samples stored in `trace`.

11.8 New functions introduced later in this guide

A few additional PyMC and ArviZ functions appear in the diagnostics, robust regression, and hierarchical modeling chapters. They follow the same pattern as everything above — declarative pieces that get attached to the model or trace, not settings to memorize in isolation.

Function	Used for	Notes
<code>pm.sample_prior_predictive(samples=500)</code>	Simulating fake data straight from the priors, before fitting	Run inside the <code>with model:</code> block, before <code>pm.sample()</code>
<code>pm.StudentT(name, nu, mu, sigma, observed)</code>	A heavier-tailed alternative likelihood, for robustness to outliers	<code>v</code> controls tail heaviness; often given its own Gamma prior
<code>shape=n_groups</code> (inside a prior)	Giving each group its own copy of a parameter in a hierarchical model	Combined with indexing, e.g. <code>beta1[region_idx]</code>
<code>az.plot_trace(trace, var_names=[...])</code>	Visually inspecting whether chains agree, iteration by iteration	Healthy traces look like flat, overlapping bands
<code>az.compare({...})</code>	Ranking multiple fitted models by estimated predictive performance	Requires <code>pm.compute_log_likelihood()</code> on each trace first