REGRESSION

FROM BEGINNER TO EXPERT

A complete, visual, application-driven course — from fitting a line through points to regularization, generalized linear models, and nonparametric methods, with a capstone on how regression powers Machine Learning, Deep Learning, and AI.

What you will be able to do

Fit and interpret linear, logistic, and generalized linear models; diagnose and fix what goes wrong; control overfitting with ridge, lasso, and cross-validation; reach beyond linearity with splines, trees, and kernels; and explain why regression is the conceptual core of supervised machine learning and the output layer of deep networks.

Format: self-paced reference & course | **Prerequisites:** basic statistics, a little linear algebra and calculus

Part of the Comprehensive Machine Learning & Deep Learning Curriculum.

Preface: how to use this book

Regression is the oldest and most useful idea in data analysis: *predict a number from other numbers, and understand the relationship between them.* Two centuries after Gauss and Legendre fit lines to astronomical data, regression is still the workhorse of statistics — and, in a deep sense, the conceptual core of machine learning. A neural network is, at heart, an enormously flexible regression function; the loss it minimizes is a regression or classification loss; the output layer that turns features into a prediction is a regression. This book builds regression from a single line through a cloud of points all the way to the methods behind modern AI.

Who this is for

This book starts from the very beginning — what it means to fit a line — and climbs to regularization, generalized linear models, splines and kernels, robust and Bayesian regression, and the regression view of deep learning. It is for the self-learner who wants both **fluency** ("I can fit the model, read the output, and trust it") and **understanding** ("I know what the coefficients mean, when the model breaks, and why these methods generalize"). Every major idea is motivated with a picture, a worked example, and a forward link to machine learning.

The central question

All of regression orbits one question and the ways we answer it:

- **The prediction problem:** given inputs $\mathbf{x}$, predict an output y. The best prediction (in squared-error terms) is the *regression function* $\mathbb{E}[Y \mid \mathbf{x}]$.
- **The estimation problem:** we never know $\mathbb{E}[Y \mid \mathbf{x}]$; we estimate it from data by choosing a model and fitting its parameters — usually by least squares or maximum likelihood.
- **The generalization problem:** a model must predict well on *new* data, not just fit the sample — the tension that drives regularization and validation.

The structure

- **Part I — Foundations** (beginner): what regression is; simple linear regression.
- **Part II — The Linear Model** (core): multiple regression and its geometry; assumptions, diagnostics, and inference; feature engineering.
- **Part III — Generalization** (core): the bias–variance tradeoff and validation; regularization (ridge, lasso, elastic net).
- **Part IV — Beyond the Linear Model** (advanced): logistic regression; generalized linear models; nonlinear and nonparametric regression.
- **Part V — Advanced & Applications**: robust, quantile, and Bayesian regression; evaluation, pitfalls, and causality; and a capstone on regression in ML/DL/AI.

How to read it

Read with a pen and a computer. When you meet a *Definition*, restate it; when you meet an *Example*, try it first; when you meet a *Try it in code* box, run it. The colored boxes recur throughout:

- **Key idea** — the one sentence to remember.

- **Intuition** — the geometric or statistical picture.
- **Common pitfall** — the mistakes that bite everyone.
- **How this powers ML / AI** — the forward link to applications.
- **Try it in code** — a hands-on check in Python (NumPy / scikit-learn).

Key idea

Hold one picture above all others: regression **draws the best curve through a cloud of data**, where "best" means smallest error. Everything else — multiple predictors, regularization, logistic and generalized models, splines, neural networks — is a variation on *which* curves are allowed and *how* we measure error. Master the line, and the rest of the subject, all the way up to deep learning, is the same idea made flexible.

Contents

PART I

Foundations

1

What Is Regression?

Level: Beginner

Regression answers a deceptively simple question: *given some inputs, what number should I predict for the output, and how are they related?* Predict a house's price from its size; a patient's risk from their labs; tomorrow's demand from today's. This chapter sets up the vocabulary — response and predictors, the regression function, signal versus noise — and draws the line between *prediction* (guess the number well) and *inference* (understand the relationship), a distinction that shapes everything that follows.

1.1 The setup: response and predictors

Definition 1.1. In regression we model a **response** (or target) y as a function of one or more **predictors** (features, covariates) $\mathbf{x} = (x_1, \dots, x_p)$ plus random noise:

$$y = f(\mathbf{x}) + \varepsilon, \qquad \mathbb{E}[\varepsilon] = 0.$$

The goal is to estimate f from data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$. When y is continuous we call it regression; when y is a category, classification — but, as we will see, the same machinery handles both.

Intuition

Think of the data as a cloud of points and f as the smooth trend running through it. The term ε is everything the predictors don't explain — measurement error, omitted causes, genuine randomness. Regression tries to recover the trend while ignoring the scatter. The art is fitting the signal f without chasing the noise ε, a tension we will formalize as the bias–variance tradeoff in Chapter 6.

1.2 The regression function is the best prediction

What is the *ideal* f? If we measure error by squared difference, the answer is a conditional expectation.

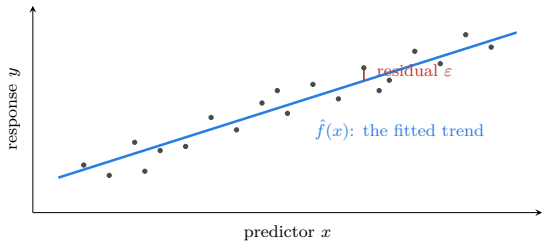
Key idea

The function that minimizes expected squared prediction error is the **regression function** — the conditional mean

$$f(\mathbf{x}) = \mathbb{E}[Y \mid \mathbf{X} = \mathbf{x}].$$

In words: *the best guess for y at a given $\mathbf{x}$ is the average of all the y's that occur there.* Every regression method — linear models, trees, neural networks — is ultimately an attempt to

approximate this conditional mean from a finite sample. Different methods just make different assumptions about its shape.



1.3 Prediction versus inference

Regression serves two distinct goals, and knowing which you are after changes how you model.

Prediction.
You only care that $\hat{y}$ is accurate on new inputs — the model can be a black box. "Will this transaction be fraudulent?" Accuracy is king; interpretability is optional. This is the machine-learning mindset.

Inference.
You want to understand the *relationship*: which predictors matter, in which direction, by how much, and with what certainty. "Does this drug lower blood pressure, controlling for age?" Here interpretable coefficients and honest uncertainty are essential. This is the classical-statistics mindset.

Common pitfall

Confusing the two goals causes real damage. A model tuned purely for predictive accuracy may have coefficients that are uninterpretable or even sign-flipped by correlated predictors — so you cannot read causal meaning off them (Chapter 4, 12). Conversely, a model built for clean inference may leave predictive accuracy on the table. Decide up front: are you trying to *predict* or to *explain*?

1.4 A first taxonomy of regression

The chapters ahead expand a single idea — approximate $\mathbb{E}[Y \mid \mathbf{x}]$ — along two axes: the *form* of f and the *type* of y.

- **Linear** ($f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{x}$): simple and multiple linear regression (Chapters 2, 3); interpretable, fast, the default.
- **Generalized linear** (a link function wraps a linear predictor): logistic for binary y, Poisson for counts (Chapters 8, 9).
- **Nonlinear / nonparametric** (flexible f): polynomials, splines, kernels, trees, Gaussian processes (Chapter 10) — and, at the extreme, neural networks (Chapter 13).

1.5 Why regression matters

Regression is the most widely used statistical method in the world, and it is the conceptual seed of supervised machine learning. Its ideas — a parametric prediction function, a loss that measures error, optimization to fit parameters, regularization to generalize — recur at every scale, from a two-column spreadsheet to a billion-parameter network.

How this powers ML / AI

Supervised machine learning *is* regression and classification at scale. A neural network is a highly flexible regression function $f_{\boldsymbol{\theta}}(\mathbf{x})$ approximating $\mathbb{E}[Y \mid \mathbf{x}]$; its final layer is literally a linear (or logistic/softmax) regression on learned features; its training minimizes a regression loss (squared error) or classification loss (cross-entropy) by gradient descent. Linear regression is the "hello world" that introduces every concept you will reuse: features, weights, a loss, fitting, overfitting, and regularization. Understand regression deeply and the rest of ML is the same melody in a richer key.

Try it in NumPy

```python
import numpy as np
from sklearn.linear_model import LinearRegression
# Data following y = 0.78 x + 1.3 + noise
rng = np.random.default_rng(0)
x = np.linspace(1, 9, 40)
y = 0.78*x + 1.3 + rng.normal(0, 0.7, size=x.size)
model = LinearRegression().fit(x.reshape(-1, 1), y)
print(model.coef_[0].round(3), model.intercept_.round(3))   # ~0.78, ~1.3
print(model.predict([[5.0]]).round(2))                      # predict y at x=5
```

Chapter summary

- Regression models a response as $y = f(\mathbf{x}) + \varepsilon$ and estimates the trend f from data while ignoring noise.
- The ideal predictor under squared error is the **regression function** $f(\mathbf{x}) = \mathbb{E}[Y \mid \mathbf{x}]$ — the conditional mean.
- Distinguish **prediction** (accurate $\hat{y}$) from **inference** (understanding the relationship); the goal shapes the model.
- Regression spans **linear**, **generalized linear**, and **nonparametric** forms — all approximating $\mathbb{E}[Y \mid \mathbf{x}]$.
- It is the conceptual core of supervised ML: a neural net is flexible regression with a regression/classification output layer.

2

Simple Linear Regression

Level: Beginner

We begin where regression itself began: fitting a straight line through points. **Simple linear regression** — one predictor, one response — is the entire subject in miniature. Every concept you will need later (a model, a loss, least-squares fitting, goodness of fit, interpretation) appears here in its clearest form. Master the line and the rest is generalization.

2.1 The model

Definition 2.1. Simple linear regression models the response as a straight-line function of one predictor plus noise:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i,$$

where β_0 is the **intercept**, β_1 the **slope**, and ε_i the error for observation i.

Intuition

The slope β_1 is the engine of interpretation: it is the expected change in y for a one-unit increase in x. The intercept β_0 is the predicted y when $x = 0$ (often an extrapolation, sometimes meaningless). Fitting the model means choosing the line — the pair (β_0, β_1) — that best matches the data. "Best" needs a definition, and that definition is least squares.

2.2 Least squares

Definition 2.2. The **residual** for point i is $e_i = y_i - \hat{y}_i$, the vertical gap between the data and the line. **Ordinary least squares** (OLS) chooses $\hat{\beta}_0, \hat{\beta}_1$ to minimize the **residual sum of squares**

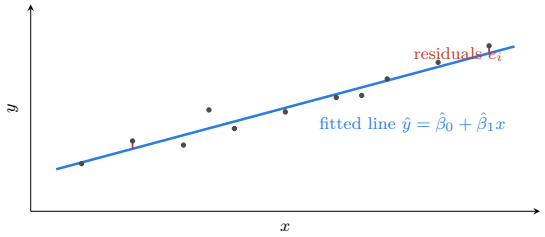
$$\mathrm{RSS} = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} \big(y_i - \beta_0 - \beta_1 x_i\big)^2.$$

Key idea

Why *squared* errors? Squaring makes all errors positive, penalizes large misses disproportionately, yields a smooth function with a unique minimum found by calculus, and — as Chapter 4 shows — corresponds to assuming Gaussian noise (so least squares is maximum likelihood). Setting the derivatives of RSS to zero gives clean closed-form estimates:

$$\hat{\beta}_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} = \frac{\mathrm{Cov}(x, y)}{\mathrm{Var}(x)}, \qquad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

Two facts worth remembering: the slope is the covariance of x and y divided by the variance of x, and the fitted line always passes through the mean point $(\bar{x}, \bar{y})$.



2.3 How good is the fit? R^2

Definition 2.3. The **coefficient of determination** measures the fraction of the response's variance explained by the model:

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}}, \qquad \text{TSS} = \sum_i (y_i - \bar{y})^2,$$

where TSS is the total variance around the mean. R^2 ranges from 0 (no better than predicting $\bar{y}$) to 1 (perfect fit).

Intuition

R^2 compares your line to the dumbest sensible model — the flat line at $\bar{y}$. It answers "how much of the wiggle in y did my predictor account for?" In simple regression, R^2 equals the squared correlation between x and y. It is intuitive but easily abused, as the pitfall warns.

Common pitfall

R^2 has sharp edges. (1) It *never decreases* when you add predictors, so a high R^2 can simply mean an overstuffed model — use adjusted R^2 or validation instead (Chapters 4, 6). (2) A high R^2 does *not* mean the model is correct, the relationship is linear, or that x *causes* y. (3) Anscombe's quartet famously shows four wildly different datasets with identical R^2 and identical fitted lines — so always *plot the data*, never trust a single number.

2.4 Interpreting and using the line

A fitted simple regression supports three uses: **describe** the association (sign and size of the slope), **predict** $\hat{y}$ at a new x, and — with the inferential tools of Chapter 4 — **test** whether the slope is reliably nonzero. Keep predictions within the range of the observed x; extrapolating a line far beyond the data is one of regression's most common and dangerous mistakes.

How this powers ML / AI

Simple linear regression is the atom from which deep learning is built. A single artificial **neuron** computes exactly $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$ — a linear regression — optionally followed by a nonlinearity. The **squared-error loss** minimized here is the MSE loss used to train regression networks; the closed-form normal equations generalize to the matrix form of Chapter 3; and when no closed form exists (as in deep nets) we minimize the same RSS by gradient descent. Understanding how slope, intercept, residuals, and R^2 behave gives

you correct intuition for the weights, biases, errors, and fit quality of any model.

Try it in NumPy

```python
import numpy as np
x = np.array([1,2,3,4,5,6,7,8,9], float)
y = np.array([2.3,3.4,3.2,4.0,4.8,5.5,6.4,7.2,8.0])
# Closed-form least squares
b1 = np.cov(x, y, bias=True)[0,1] / np.var(x)
b0 = y.mean() - b1*x.mean()
yhat = b0 + b1*x
R2 = 1 - ((y-yhat)**2).sum() / ((y-y.mean())**2).sum()
print(round(b0,3), round(b1,3), round(R2,4))     # intercept, slope, R^2
print(round(np.corrcoef(x,y)[0,1]**2, 4))         # equals R^2 in simple regression
```

Chapter summary

- Simple linear regression fits $y = \beta_0 + \beta_1 x + \varepsilon$; the **slope** is the change in y per unit x.
- **Least squares** minimizes RSS $= \sum e_i^2$, giving closed forms $\hat{\beta}_1 = \mathrm{Cov}(x,y)/\mathrm{Var}(x)$, $\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$.
- The line passes through $(\bar{x}, \bar{y})$; squared error corresponds to a Gaussian-noise / maximum-likelihood assumption.
- $R^2 = 1 - \mathrm{RSS}/\mathrm{TSS}$ is the fraction of variance explained — intuitive but easily abused; always plot the data (Anscombe).
- A neuron is a linear regression; this chapter's loss and fitting recur throughout ML.

PART II

The Linear Model

3

Multiple Linear Regression and Its Geometry

Level: Core

Real problems have many predictors at once: price depends on size *and* location *and* age. **Multiple linear regression** extends the line to a plane, then a hyperplane, and — crucially — introduces the matrix algebra and geometric picture that unify all of linear modeling. This chapter is the structural heart of the book: the normal equations and the projection view here reappear in regularization, GLMs, and the output layer of every neural network.

3.1 The model in matrix form

Definition 3.1. With p predictors, multiple linear regression models

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i.$$

Stacking observations into a **design matrix** $\mathbf{X} \in \mathbb{R}^{n \times (p+1)}$ (a leading column of ones absorbs the intercept), a response vector $\mathbf{y} \in \mathbb{R}^n$, and coefficients $\boldsymbol{\beta} \in \mathbb{R}^{p+1}$, the whole dataset becomes

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}.$$

Intuition

Each *row* of $\mathbf{X}$ is one observation; each *column* is one feature across all observations. The product $\mathbf{X}\boldsymbol{\beta}$ forms, for every row, the weighted sum of features — the prediction. Compressing n scalar equations into one matrix equation is not just shorthand: it lets us solve for all coefficients at once and exposes the geometry that makes least squares intuitive.

3.2 The normal equations

Key idea

Minimizing $\mathrm{RSS}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2$ leads, by setting the gradient to zero, to the **normal equations**

$$\mathbf{X}^\top \mathbf{X}\,\hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{y} \quad \Longrightarrow \quad \boxed{\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}}$$

(whenever $\mathbf{X}^\top \mathbf{X}$ is invertible). This single formula is the workhorse of classical regression — it generalizes the slope/intercept formulas of Chapter 2 to any number of predictors.

Intuition

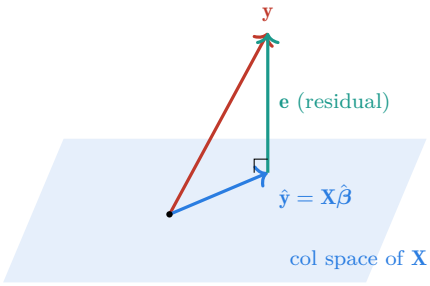
The matrix $\mathbf{X}^{+} = (\mathbf{X}^{\top}\mathbf{X})^{-1}\mathbf{X}^{\top}$ is the **Moore–Penrose pseudoinverse**. If $\mathbf{X}$ were square and invertible we would just solve $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ exactly; but $\mathbf{X}$ is tall ($n > p$), the system is overdetermined, and no exact solution exists. The pseudoinverse delivers the *least-squares* solution: the $\hat{\boldsymbol{\beta}}$ whose predictions come closest to $\mathbf{y}$.

3.3 The geometry: regression is projection

Here is the picture that makes least squares click.

Key idea

The fitted values $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$ are the **orthogonal projection** of $\mathbf{y}$ onto the *column space* of $\mathbf{X}$ — the subspace of all vectors reachable as $\mathbf{X}\boldsymbol{\beta}$. Least squares finds the point in that subspace closest to $\mathbf{y}$; the residual $\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}}$ is the leftover, and it is *perpendicular* to every predictor: $\mathbf{X}^{\top}\mathbf{e} = \mathbf{0}$. That orthogonality *is* the normal equations.



Intuition

Drop a perpendicular from the tip of $\mathbf{y}$ to the plane spanned by the predictors; the foot of that perpendicular is $\hat{\mathbf{y}}$. "Least squares" literally means "shortest residual," and the shortest connection from a point to a plane is the perpendicular one. The matrix $\mathbf{H} = \mathbf{X}(\mathbf{X}^{\top}\mathbf{X})^{-1}\mathbf{X}^{\top}$ that performs this projection ($\hat{\mathbf{y}} = \mathbf{H}\mathbf{y}$) is called the **hat matrix** because it "puts the hat on $\mathbf{y}$."

3.4 Interpreting coefficients: "holding others fixed"

Definition 3.2. In multiple regression, β_j is the expected change in y for a one-unit increase in x_j *while all other predictors are held constant.* This *ceteris paribus* interpretation is what makes multiple regression a tool for adjustment and control.

Common pitfall

The phrase "holding others fixed" is treacherous when predictors are correlated (**multicollinearity**). If x_1 and x_2 move together in the data, the model cannot cleanly separate their effects: coefficients become unstable, have huge standard errors, and can even flip sign. A coefficient is *not* the raw association between x_j and y — it is the association *after removing what the other predictors explain.* This is also why a predictor can look important alone yet insignificant in the full model, and vice versa. We diagnose this with the variance inflation factor in Chapter 4 and fix it with regularization in Chapter 7.

3.5 When $\mathbf{X}^\top\mathbf{X}$ is not invertible

If predictors are linearly dependent (e.g. redundant columns, or $p > n$ as in genomics and modern ML), $\mathbf{X}^\top\mathbf{X}$ is singular and the inverse does not exist — there are infinitely many least-squares solutions. Two remedies dominate: use the pseudoinverse (which picks the minimum-norm solution) or add a penalty that restores invertibility — exactly ridge regression, $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$ (Chapter 7).

How this powers ML / AI

This chapter *is* the linear layer of a neural network. A `Dense`/`Linear` layer computes $\mathbf{XW}$ — the identical matrix multiply, with the weight matrix playing the role of $\boldsymbol{\beta}$ and many outputs at once. The final layer of a regression network is multiple linear regression on *learned* features; the projection geometry explains why the network's last hidden layer is a learned feature space in which the target is (approximately) a linear function. And the singular-$\mathbf{X}^\top\mathbf{X}$ problem is ubiquitous in deep learning, where $p \gg n$ — which is precisely why weight decay (ridge) is a default. Modern libraries avoid forming $(\mathbf{X}^\top\mathbf{X})^{-1}$ explicitly, solving via QR or SVD for numerical stability, but the math is this.

Try it in NumPy

```python
import numpy as np
rng = np.random.default_rng(0)
n = 200
X = np.column_stack([np.ones(n), rng.normal(size=n), rng.normal(size=n)])
beta_true = np.array([1.0, 2.0, -0.5])
y = X @ beta_true + rng.normal(0, 0.5, size=n)
# Normal equations (use lstsq in practice for stability, not the explicit inverse)
beta_hat = np.linalg.solve(X.T @ X, X.T @ y)
print(beta_hat.round(3))                          # ~[1.0, 2.0, -0.5]
resid = y - X @ beta_hat
print(np.allclose(X.T @ resid, 0, atol=1e-9))  # residual orthogonal to columns -> True
```

Chapter summary

- Multiple regression writes the whole dataset as $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ with a design matrix $\mathbf{X}$.
- The **normal equations** give $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$ — the multivariable least-squares solution.
- Geometrically, least squares **projects** $\mathbf{y}$ onto the column space of $\mathbf{X}$; the residual is orthogonal to all predictors ($\mathbf{X}^\top\mathbf{e} = \mathbf{0}$).
- Each β_j is the effect of x_j *holding others fixed*; correlated predictors (**multicollinearity**) make this unstable.
- Singular $\mathbf{X}^\top\mathbf{X}$ (e.g. $p > n$) needs the pseudoinverse or a ridge penalty — the same linear-layer math used throughout deep learning.

4

Assumptions, Diagnostics, and Inference

Level: Core

A fitted model always produces coefficients — but whether you can *trust* them, predict with them, or attach uncertainty to them depends on assumptions. This chapter states what OLS assumes, shows how to check it with residual plots, and develops the inference machinery (standard errors, t- and F-tests, confidence and prediction intervals) that turns point estimates into honest conclusions.

4.1 The classical assumptions

Definition 4.1. The standard linear-regression assumptions, often abbreviated **LINE**, are:

1. **Linearity:** $\mathbb{E}[y \mid \mathbf{x}] = \boldsymbol{\beta}^\top \mathbf{x}$ — the mean is linear in the predictors.
2. **Independence:** the errors ε_i are independent across observations.
3. **Normality:** ε_i are (approximately) Gaussian — needed for exact small-sample inference.
4. **Equal variance (homoscedasticity):** $\mathrm{Var}(\varepsilon_i) = \sigma^2$ is constant.

Key idea

The **Gauss–Markov theorem** says that under linearity, independence, zero-mean errors, and constant variance (normality *not* required), OLS is **BLUE** — the Best Linear Unbiased Estimator, meaning it has the smallest variance among all unbiased estimators that are linear in $\mathbf{y}$. This is why least squares is the default: it is optimal in a precise sense, for free, without distributional assumptions. Normality is needed only for exact t/F inference; for large n the Central Limit Theorem rescues us.

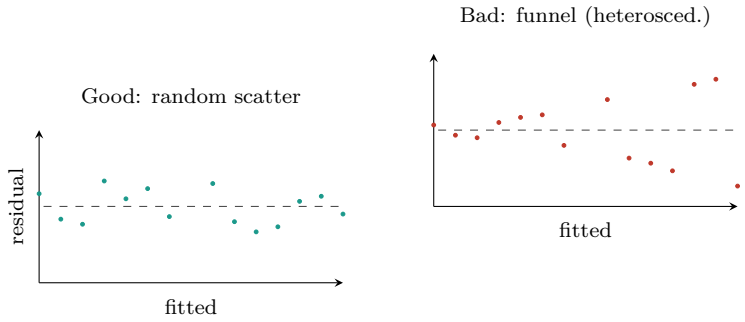
4.2 Diagnostics: read the residuals

The residuals are your window into assumption violations. A handful of plots catches most problems.

Residuals vs. fitted.
Should be a structureless cloud around zero. A *curve* signals missing nonlinearity (transform or add terms); a *funnel* signals heteroscedasticity.

Q–Q plot.
Residual quantiles vs. normal quantiles; points off the diagonal in the tails reveal non-normality or outliers.

Scale–location & leverage.
Detect non-constant spread and high-leverage / high-influence points (Cook's distance).



4.3 Inference: standard errors and tests

Definition 4.2. Under the assumptions, the coefficient estimator is Gaussian with covariance

$$\widehat{\mathrm{Cov}}(\hat{\boldsymbol{\beta}}) = \hat{\sigma}^2(\mathbf{X}^\top\mathbf{X})^{-1}, \qquad \hat{\sigma}^2 = \frac{\mathrm{RSS}}{n-p-1}.$$

The square roots of the diagonal are the **standard errors** $\mathrm{SE}(\hat{\beta}_j)$.

These give the standard tests:

- ***t*-test** for a single coefficient: $t_j = \hat{\beta}_j/\mathrm{SE}(\hat{\beta}_j)$ tests $H_0 : \beta_j = 0$. The p-value answers "could this effect plausibly be zero?"
- ***F*-test** for the whole model (or a group of coefficients): tests whether the predictors *jointly* explain variance beyond the intercept.
- **Confidence interval:** $\hat{\beta}_j \pm t^* \,\mathrm{SE}(\hat{\beta}_j)$ — plausible values for the true effect.

Intuition

A coefficient is "significant" when it is large *relative to its uncertainty*. Doubling the sample size shrinks standard errors (roughly as $1/\sqrt{n}$), so with enough data even tiny effects become statistically significant — which is why *effect size* matters more than the p-value alone.

4.4 Confidence vs. prediction intervals

Common pitfall

Do not confuse a **confidence interval** for the mean response with a **prediction interval** for a new observation. The confidence interval captures uncertainty about the *average* y at a given $\mathbf{x}$ (only estimation error). The prediction interval also includes the irreducible noise σ^2 of a single draw, so it is *always wider*. Reporting a narrow confidence interval when a user actually needs a prediction interval drastically understates real-world uncertainty.

4.5 Multicollinearity and the VIF

Definition 4.3. The **variance inflation factor** for predictor j is $\text{VIF}_j = 1/(1 - R_j^2)$, where R_j^2 is from regressing x_j on the other predictors. It measures how much collinearity inflates $\text{Var}(\hat{\beta}_j)$. Rules of thumb: VIF > 5 is concerning, > 10 is severe.

How this powers ML / AI

The assumptions explain *when* a model's coefficients are trustworthy — directly relevant to interpretable / explainable ML and to causal inference. Heteroscedasticity motivates **weighted least squares** and, in deep learning, *heteroscedastic* loss heads that predict both a mean and a variance (aleatoric uncertainty). Multicollinearity is why correlated features destabilize linear models and why **regularization** (Chapter 7) and decorrelating transforms like PCA are everyday tools. And the covariance $\hat{\sigma}^2(\mathbf{X}^\top\mathbf{X})^{-1}$ is the classical ancestor of the Laplace approximation used for uncertainty in Bayesian neural networks.

Try it in NumPy

```
import numpy as np, statsmodels.api as sm
rng = np.random.default_rng(1)
n = 100
x1 = rng.normal(size=n); x2 = rng.normal(size=n)
y = 1 + 2*x1 - 0.5*x2 + rng.normal(0, 1, n)
X = sm.add_constant(np.column_stack([x1, x2]))
fit = sm.OLS(y, X).fit()
print(fit.summary())            # coefficients, SEs, t-stats, p-values, R^2, F-stat
print(fit.conf_int())           # 95% confidence intervals for each beta
```

Chapter summary

- OLS assumes **LINE**: linearity, independence, normality (for exact inference), equal variance.
- **Gauss–Markov**: OLS is BLUE under the mean/variance assumptions — normality is needed only for small-sample t/F tests.
- Diagnose with residual plots: curves → nonlinearity, funnels → heteroscedasticity, Q–Q tails → non-normality/outliers.
- Standard errors from $\hat{\sigma}^2(\mathbf{X}^\top\mathbf{X})^{-1}$ drive t-tests, F-tests, and confidence intervals; **prediction** intervals are wider than **confidence** intervals.
- The **VIF** flags multicollinearity, which destabilizes coefficients and motivates regularization.

5

Feature Engineering and Model Building

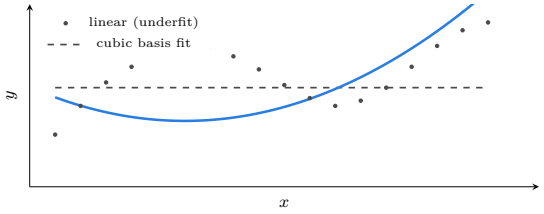
Level: Core

Linear regression is linear *in the parameters*, not necessarily in the raw inputs. That loophole is enormous: by transforming and combining predictors we can model curves, interactions, categories, and saturating effects — all while keeping the simple machinery of least squares. This chapter is about **feature engineering**: building the columns of $\mathbf{X}$ so that a linear model can capture nonlinear reality.

5.1 Polynomial and basis-function regression

Key idea

Replace x with a vector of **basis functions** $\phi(x) = (1, x, x^2, \dots)$ and the model $y = \boldsymbol{\beta}^\top \phi(x) + \varepsilon$ is still linear in $\boldsymbol{\beta}$ — so OLS solves it unchanged. The fitted curve is nonlinear in x but the *estimation* is ordinary linear regression on the expanded design matrix. This "feature map" trick is the bridge from lines to arbitrary curves, and (Chapter 10) the seed of kernels and splines.



5.2 Categorical predictors: dummy coding

Definition 5.1. A categorical variable with k levels enters the model as $k-1$ **dummy** (indicator) variables, each 0/1. One level is the **reference**; each dummy's coefficient is the difference in mean response *relative to the reference*, holding other predictors fixed.

Common pitfall

Including a dummy for *all* k levels *and* an intercept makes the columns sum to the intercept column — perfect collinearity, a singular $\mathbf{X}^\top\mathbf{X}$. This is the **dummy variable trap**; always drop one level (or the intercept). Note that one-hot encoding in ML keeps all k columns

precisely because regularization (or no explicit intercept) sidesteps the singularity.

5.3 Interactions

Definition 5.2. An **interaction** term x_1x_2 lets the effect of one predictor depend on another: in $y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_1x_2$, the slope of y in x_1 is $\beta_1 + \beta_3x_2$. Without the product term, the model forces effects to be purely additive.

Intuition

"Does the effect of dose depend on age?" is an interaction question. A purely additive model answers "no" by assumption; adding x_1x_2 lets the data decide. Deep networks discover such interactions automatically through their hidden layers — a feed-forward net is, in part, an interaction-and-transformation machine — which is one reason they need less manual feature engineering.

5.4 Transformations and scaling

- **Log transforms** tame right-skew and turn multiplicative relationships additive; with a log response, coefficients read as approximate *percentage* changes (a log–log model gives elasticities).
- **Standardization** ($z = (x - \bar{x})/s$) puts predictors on a common scale so coefficients are comparable and — critically — so regularization (Chapter 7) and gradient descent behave well.
- **Box–Cox / Yeo–Johnson** choose a power transform to stabilize variance and improve normality.

Common pitfall

Data leakage via preprocessing is one of the most common and silent errors. Compute scaling parameters, imputation values, and transforms on the *training fold only*, then apply them to validation/test data. Fitting a scaler on the whole dataset before splitting leaks information about the test set into training and inflates your measured performance. Use a `Pipeline` so the transform is refit inside every cross-validation fold (Chapter 6).

How this powers ML / AI

Feature engineering is the original "representation learning." Classical ML lived or died by hand-crafted features — polynomials, interactions, domain transforms — fed to a linear model. The deep-learning revolution automated this: hidden layers *learn* the feature map $\phi(\mathbf{x})$ that earlier we built by hand, and the final linear layer is the regression on top. The kernel trick (Chapter 10) is the same idea taken to an implicit infinite-dimensional ϕ. Whether features are engineered or learned, the downstream model is still the linear regression of this book — which is why these fundamentals transfer directly.

Try it in NumPy

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
```

```python
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
rng = np.random.default_rng(0)
x = np.linspace(-3, 3, 80).reshape(-1, 1)
y = 0.5*x.ravel()**2 - x.ravel() + rng.normal(0, 0.4, 80)
# Pipeline keeps scaling + basis expansion leak-free inside CV
model = make_pipeline(StandardScaler(),
                      PolynomialFeatures(degree=2, include_bias=False),
                      LinearRegression()).fit(x, y)
print(model.score(x, y).round(3))   # R^2 of the quadratic-feature fit
```

Chapter summary

- Linear regression is linear in *parameters*: basis functions $\phi(x)$ let it fit curves while OLS stays unchanged.
- Categorical predictors use $k-1$ **dummies**; including all k with an intercept is the **dummy variable trap**.
- **Interactions** ($x_1 x_2$) let one predictor's effect depend on another; networks learn these automatically.
- Log transforms, **standardization**, and Box–Cox reshape features; scaling is essential for regularization and gradient descent.
- Fit all preprocessing on training data only — preprocessing on the full set causes **data leakage**. Hidden layers automate feature engineering.

PART III

Generalization

6

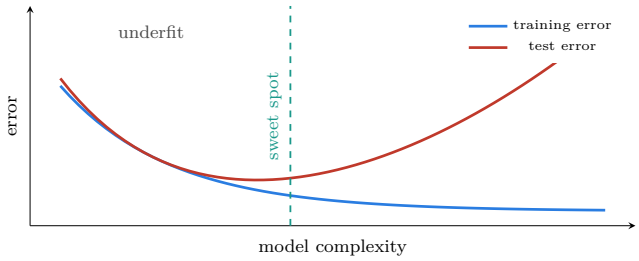
The Bias–Variance Tradeoff and Model Validation

Level: Core

A model that fits the training data perfectly is often useless on new data. This chapter explains why — the **bias–variance tradeoff**, the most important concept in all of predictive modeling — and gives the tools to measure generalization honestly: train/test splits, cross-validation, and information criteria. These ideas govern every choice from polynomial degree to neural-network size.

6.1 Overfitting and underfitting

Definition 6.1. A model **underfits** when it is too simple to capture the true pattern (high error on both training and new data). It **overfits** when it is so flexible that it memorizes noise in the training data (low training error, high error on new data). The goal is the sweet spot between them.



6.2 Decomposing the error

Key idea

For squared-error loss, the expected test error at a point decomposes exactly into three pieces:

$$\mathbb{E}\big[(y-\hat{f}(\mathbf{x}))^2\big] = \underbrace{\big(\mathrm{Bias}[\hat{f}(\mathbf{x})]\big)^2}_{\text{too rigid}} + \underbrace{\mathrm{Var}[\hat{f}(\mathbf{x})]}_{\text{too sensitive}} + \underbrace{\sigma^2}_{\text{irreducible}} .$$

Bias is error from wrong assumptions (a line can't fit a curve). **Variance** is sensitivity to the particular training sample (a wiggly curve changes wildly with new data). **Irreducible noise** σ^2 is the floor no model can beat. Simplifying lowers variance but raises bias; flexibility does the reverse — you cannot drive both to zero, so you *balance* them.

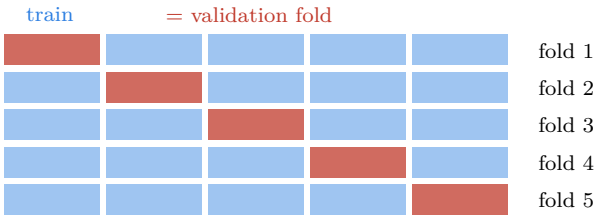
Intuition

Picture hitting a target with darts. *High bias, low variance:* tight cluster, but off-center (consistent and wrong). *Low bias, high variance:* centered on average, but scattered everywhere (right on average, wrong each time). Good prediction needs both centered *and* tight — and since data is finite, getting there means deliberately accepting a little bias (via regularization) to buy a large reduction in variance.

6.3 Honest evaluation: train/test and cross-validation

Definition 6.2. Estimate generalization on data the model has *not* seen.

- **Train/validation/test split:** fit on train, tune on validation, report once on test.
- **k-fold cross-validation:** split data into k folds; train on $k-1$, validate on the held-out fold, rotate, and average. Uses all data for both roles and gives a lower-variance performance estimate.
- **Leave-one-out (LOOCV):** the extreme $k = n$; for linear models it has a remarkable closed form via the hat-matrix diagonal (no refitting needed).



6.4 Information criteria

For models fit by maximum likelihood, **AIC** and **BIC** estimate out-of-sample quality without a held-out set by penalizing complexity:

$$\mathrm{AIC} = -2\ln\hat{L} + 2k, \qquad \mathrm{BIC} = -2\ln\hat{L} + k\ln n,$$

where k is the number of parameters. Lower is better; BIC penalizes complexity more harshly (and favors smaller models) as n grows. They are fast surrogates for cross-validation when refitting is expensive.

Common pitfall

The cardinal sin of evaluation is letting the test set influence the model. If you tune hyperparameters on the test set, peek at it repeatedly, or preprocess before splitting, your reported error is optimistic and your model will disappoint in production. Lock the test set away; do all selection with cross-validation on the training data. Also beware that with non-IID data (time series, grouped/clustered data) ordinary k-fold leaks — use time-series or grouped splits instead.

How this powers ML / AI

The bias–variance tradeoff is the organizing principle of all of machine learning: model capacity, regularization strength, tree depth, network width, dropout, and early stopping are all knobs on this single dial. Cross-validation is the universal protocol for hyperparameter tuning. Intriguingly, very large neural networks exhibit **double descent**: past the

interpolation threshold, test error can fall *again* even as the classical curve says it should rise — a refinement, not a refutation, of this framework, and an active research frontier. The intuition you build here remains the foundation for reasoning about every model's generalization.

Try it in NumPy

```python
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
rng = np.random.default_rng(0)
x = np.sort(rng.uniform(-3, 3, 60)).reshape(-1, 1)
y = np.sin(x).ravel() + rng.normal(0, 0.2, 60)
for d in [1, 3, 9, 15]:
    model = make_pipeline(PolynomialFeatures(d), LinearRegression())
    mse = -cross_val_score(model, x, y, cv=5,
                           scoring="neg_mean_squared_error").mean()
    print(f"degree {d:2d}: CV MSE = {mse:.3f}")   # U-shaped: best at
    moderate degree
```

Chapter summary

- **Underfitting** = too rigid (high bias); **overfitting** = memorizing noise (high variance). Aim for the middle.
- Expected squared error = **bias**^{2} + **variance** + irreducible noise σ^2; you trade bias against variance, never eliminate both.
- Measure generalization on unseen data: train/test splits and **k-fold cross-validation**; LOOCV has a closed form for linear models.
- **AIC/BIC** approximate out-of-sample quality by penalizing parameters; BIC penalizes harder.
- Never let the test set leak into modeling; this tradeoff governs every ML capacity knob (with double descent as a modern twist).

7

Regularization: Ridge, Lasso, and Elastic Net

Level: Advanced

When predictors are many or correlated, ordinary least squares overfits: coefficients explode, variance soars, and predictions on new data suffer. **Regularization** fixes this by adding a penalty that discourages large coefficients — deliberately introducing a little bias to slash variance. Ridge, lasso, and elastic net are the three canonical penalties, and weight decay — the default in deep learning — is exactly ridge.

7.1 The idea: penalize complexity

Key idea

Instead of minimizing fit alone, minimize fit *plus* a penalty on coefficient size:

$$\min_{\boldsymbol{\beta}} \underbrace{\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2}_{\text{fit}} + \lambda \underbrace{\Omega(\boldsymbol{\beta})}_{\text{penalty}}.$$

The tuning parameter $\lambda \geq 0$ sets the strength: $\lambda = 0$ recovers OLS; $\lambda \to \infty$ shrinks all coefficients to zero. Choosing λ *is* navigating the bias–variance tradeoff of Chapter 6, and it is chosen by cross-validation.

7.2 Ridge (L_2)

Definition 7.1. Ridge regression penalizes the squared L_2 norm, $\Omega(\boldsymbol{\beta}) = \sum_j \beta_j^2$, and has a closed form:

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}.$$

Intuition

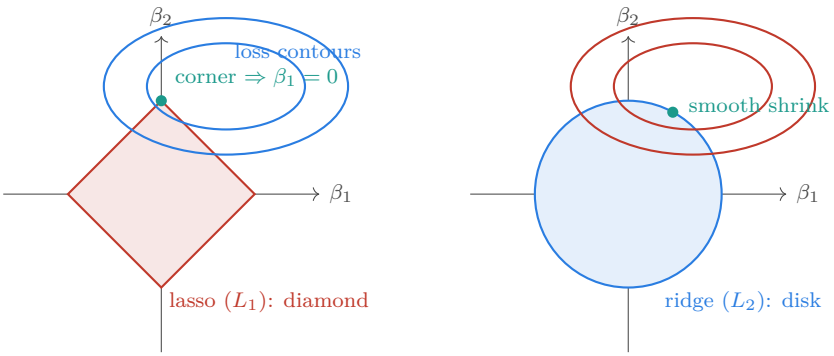
The added $\lambda\mathbf{I}$ literally lifts the diagonal of $\mathbf{X}^\top\mathbf{X}$, guaranteeing invertibility even when $p > n$ or predictors are collinear — ridge *always* has a unique solution. It shrinks coefficients smoothly toward zero (never exactly to zero), sharing weight among correlated predictors rather than arbitrarily picking one. Probabilistically, ridge is the **MAP estimate** under a Gaussian prior $\beta_j \sim \mathcal{N}(0, \tau^2)$ — regularization is a prior belief that effects are small.

7.3 Lasso (L_1)

Definition 7.2. The **lasso** penalizes the L_1 norm, $\Omega(\boldsymbol{\beta}) = \sum_j |\beta_j|$. It has no closed form but is convex and solved efficiently (coordinate descent, LARS).

Key idea

The lasso's signature property is **sparsity**: it drives some coefficients *exactly* to zero, performing variable selection and fitting simultaneously. The geometry is the reason — the L_1 constraint region is a diamond with sharp corners on the axes, so the elliptical contours of the loss tend to first touch it at a corner, where some coordinates are zero. Lasso is the MAP estimate under a Laplace prior.



7.4 Elastic net

Definition 7.3. The **elastic net** blends both penalties, $\Omega(\boldsymbol{\beta}) = \alpha \sum_j |\beta_j| + (1-\alpha) \sum_j \beta_j^2$, combining lasso's sparsity with ridge's stability.

Intuition

Pure lasso struggles when predictors are highly correlated — it arbitrarily keeps one and zeros the rest, and it can select at most n variables when $p > n$. Elastic net's ridge component encourages correlated predictors to enter or leave *together* (the "grouping effect"), making it the practical default for wide, correlated data such as genomics and text.

Common pitfall

Always **standardize** predictors before regularizing. The penalty acts on coefficient magnitudes, so a predictor measured in small units (large coefficient) is penalized more than the same predictor in large units — regularization is not scale-invariant. Also, conventionally the intercept is left *unpenalized*; you do not want to shrink the baseline level of the response. Most libraries handle both automatically, but verify.

How this powers ML / AI

Regularization is everywhere in deep learning, often under the name **weight decay** — which is precisely the ridge (L_2) penalty added to the loss, and AdamW made it a default for training Transformers. L_1 penalties induce sparse networks for compression and pruning.

The Bayesian reading — penalty = prior — connects directly to Bayesian neural networks and variational inference. Even techniques without an explicit penalty term, like **early stopping** and **dropout**, are understood as implicit regularizers that control the same bias–variance balance. The single equation "loss $+\,\lambda\cdot$ penalty" is the template for nearly every regularized objective in modern AI.

Try it in NumPy

```
import numpy as np
from sklearn.linear_model import RidgeCV, LassoCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
rng = np.random.default_rng(0)
n, p = 100, 30
X = rng.normal(size=(n, p))
beta = np.zeros(p); beta[:5] = [3, -2, 1.5, 0, 2]   # only a few truly nonzero
y = X @ beta + rng.normal(0, 1, n)
ridge = make_pipeline(StandardScaler(), RidgeCV(alphas=np.logspace(-2,2,50))).fit(X, y)
lasso = make_pipeline(StandardScaler(), LassoCV(cv=5)).fit(X, y)
print("lasso zeros:", int(np.sum(np.abs(lasso[-1].coef_) < 1e-6)), "of", p)  # sparsity
```

Chapter summary

- Regularization minimizes **fit** $+\,\lambda\cdot$ **penalty**, trading a little bias for much less variance; tune λ by cross-validation.
- **Ridge** (L_2) shrinks smoothly, always invertible, shares weight among correlated predictors; it is weight decay and a Gaussian-prior MAP.
- **Lasso** (L_1) yields **sparse** solutions (exact zeros) — automatic variable selection; Laplace-prior MAP.
- **Elastic net** blends both, handling correlated, high-dimensional data with a grouping effect.
- Standardize first; leave the intercept unpenalized. "Loss + penalty" is the template for regularization across all of ML.

PART IV

Beyond the Linear Model

8

Logistic Regression and Classification

Level: Advanced

What if the response is a category — spam or not, disease or healthy, click or no click? Fitting a line to 0/1 labels gives nonsensical probabilities outside $[0, 1]$. **Logistic regression** fixes this elegantly: it runs a linear model through a squashing function so the output is a valid probability. It is the most important classification method in statistics and — not coincidentally — the exact form of a neuron with a sigmoid activation and the building block of every classification network.

8.1 From linear to logistic

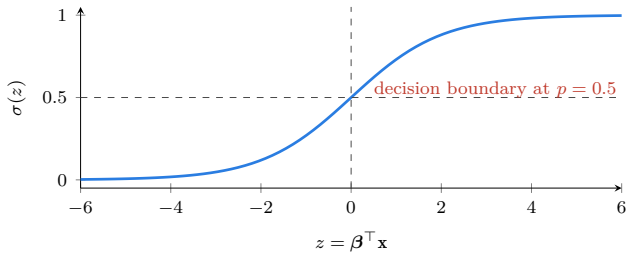
Key idea

Logistic regression models the *probability* of the positive class by passing a linear predictor through the **sigmoid** (logistic) function:

$$p(\mathbf{x}) = \Pr(y = 1 \mid \mathbf{x}) = \sigma(\boldsymbol{\beta}^\top \mathbf{x}), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

The sigmoid maps any real number to $(0, 1)$, so predictions are always valid probabilities. Equivalently, the model is linear in the **log-odds**:

$$\operatorname{logit}(p) = \ln \frac{p}{1 - p} = \boldsymbol{\beta}^\top \mathbf{x}.$$



Intuition

The **odds** $p/(1-p)$ turn a probability into a ratio ("3 to 1"); the log-odds stretch it to the whole real line so a linear model fits naturally. A coefficient β_j means: a one-unit increase in x_j adds β_j to the log-odds, i.e. *multiplies the odds by* e^{β_j}. The decision boundary $\boldsymbol{\beta}^\top \mathbf{x} = 0$ (where $p = 0.5$) is a hyperplane — logistic regression is a *linear* classifier, even though its probabilities curve.

8.2 Fitting: maximum likelihood and cross-entropy

Definition 8.1. There is no least-squares closed form. Instead we maximize the **likelihood** of the observed labels, equivalently minimizing the **binary cross-entropy** (log loss):

$$\mathcal{L}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \Big[y_i \ln p(\mathbf{x}_i) + (1 - y_i) \ln \big(1 - p(\mathbf{x}_i)\big) \Big].$$

Intuition

Cross-entropy punishes confident wrong predictions brutally: predicting $p = 0.99$ for a true 0 incurs near-infinite loss, while a hedged $p = 0.5$ is mild. This is the negative log-likelihood under a Bernoulli model, it is convex (a unique optimum), and there is no closed form — so we optimize numerically by Newton's method (IRLS) or gradient descent. That last fact is the bridge to neural networks, which minimize this *exact* loss by gradient descent.

8.3 Multiclass: softmax regression

Definition 8.2. For $K > 2$ classes, **softmax** (multinomial logistic) regression generalizes the sigmoid:

$$\Pr(y = k \mid \mathbf{x}) = \frac{e^{\boldsymbol{\beta}_k^\top \mathbf{x}}}{\sum_{j=1}^{K} e^{\boldsymbol{\beta}_j^\top \mathbf{x}}}.$$

Each class gets its own coefficient vector; softmax normalizes the scores into a probability distribution that sums to one.

Common pitfall

Several traps recur. (1) **Perfect separation**: if a feature perfectly splits the classes, the MLE coefficients diverge to $\pm\infty$ — use regularization (a penalty, equivalently a prior) to keep them finite. (2) **Threshold $\neq$ probability**: the default 0.5 cutoff is rarely optimal under class imbalance or unequal error costs; choose the threshold from a precision–recall or ROC analysis. (3) **Accuracy lies** on imbalanced data — a 99%-negative dataset is "99% accurate" by always predicting negative; use AUC, F1, or calibrated probabilities instead (Chapter 12).

How this powers ML / AI

Logistic regression *is* a single sigmoid neuron, and softmax regression is exactly the final layer of essentially every classification network — from image classifiers to the next-token predictor of a large language model, which is a softmax over the vocabulary. The cross-entropy loss minimized here is *the* classification loss of deep learning. So a neural classifier is best understood as logistic/softmax regression performed on *learned* features rather than raw inputs: the hidden layers learn a representation in which the classes are linearly separable, and the output layer is this chapter. Master logistic regression and you understand the head of every deep classifier.

Try it in NumPy

```python
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=400, n_features=5, n_informative
    =3, random_state=0)
clf = LogisticRegression().fit(X, y)
p = clf.predict_proba(X[:3])[:, 1]
print("P(y=1):", p.round(3))
print("odds multiplier per unit x_j:", np.exp(clf.coef_[0]).round(3))
      # exp(beta)
```

Chapter summary

- Logistic regression models $\Pr(y = 1 \mid \mathbf{x}) = \sigma(\boldsymbol{\beta}^\top \mathbf{x})$; it is linear in the **log-odds**, with e^{β_j} the odds multiplier.
- The decision boundary is a hyperplane — a *linear* classifier with curved probabilities.
- Fit by **maximum likelihood** = minimizing **cross-entropy** (log loss); convex, no closed form, solved by IRLS / gradient descent.
- **Softmax** regression extends it to K classes; watch for perfect separation, threshold choice, and imbalance.
- It is a sigmoid neuron and the output layer of every deep classifier; cross-entropy is the classification loss of all of ML.

9

Generalized Linear Models

Level: Advanced

Linear regression assumes Gaussian, constant-variance noise; logistic regression handles binary outcomes. **Generalized linear models** (GLMs) reveal these as two members of one family, unifying regression for continuous, binary, count, and positive responses under a single, elegant template. Understanding the template — a random component, a linear predictor, and a link function — lets you choose the right model for any response type, and clarifies what a neural network's output layer and loss really are.

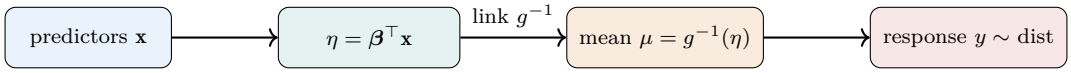
9.1 The three ingredients

Definition 9.1. A GLM has three parts:

1. **Random component:** the response follows a distribution from the *exponential family* (Gaussian, Bernoulli, Poisson, Gamma, …).
2. **Linear predictor:** $\eta = \boldsymbol{\beta}^\top \mathbf{x}$, the familiar linear combination.
3. **Link function** g: connects the mean to the linear predictor, $g(\mu) = \eta$, i.e. $\mu = g^{-1}(\boldsymbol{\beta}^\top \mathbf{x})$.

Key idea

The **link function** is the key generalization. Ordinary regression uses the *identity* link ($\mu = \boldsymbol{\beta}^\top \mathbf{x}$); logistic regression uses the *logit* link ($\ln \frac{\mu}{1-\mu} = \boldsymbol{\beta}^\top \mathbf{x}$); Poisson regression uses the *log* link ($\ln \mu = \boldsymbol{\beta}^\top \mathbf{x}$). One framework, one fitting algorithm (iteratively reweighted least squares), many response types — you just pick the distribution and link that match your data.



9.2 The common GLMs

Response type	Distribution	Link	Model
Continuous	Gaussian	identity	linear regression
Binary / proportion	Bernoulli / Binomial	logit	logistic regression
Count	Poisson	log	Poisson regression
Count (overdispersed)	Negative binomial	log	NB regression
Positive, skewed	Gamma	log / inverse	Gamma regression

Intuition

Poisson regression is the everyday count model: the log link guarantees a positive mean and turns multiplicative effects additive, so e^{β_j} is a *rate ratio* — "each unit of x_j multiplies the expected count by e^{β_j}." Use it for events per unit time/space: website visits, insurance claims, disease incidence. Watch for **overdispersion** (variance exceeding the mean), where the negative binomial is the fix.

9.3 Why the exponential family?

Definition 9.2. A distribution is in the **exponential family** if its density can be written

$$f(y;\theta) = h(y)\exp\big(\theta\, y - A(\theta)\big),$$

for a natural parameter θ and log-partition function $A(\theta)$. The Gaussian, Bernoulli, Poisson, Gamma, and many others all fit this mold.

This shared structure is what makes one fitting algorithm work for all GLMs: the likelihood is concave, the **canonical link** arises naturally, and maximum likelihood reduces to iteratively reweighted least squares — weighted versions of the normal equations from Chapter 3.

Common pitfall

Two frequent mistakes. (1) Using ordinary linear regression on counts or probabilities — it predicts impossible negative counts or probabilities outside $[0, 1]$ and assumes the wrong (constant) variance; reach for the matching GLM instead. (2) Ignoring **overdispersion** in Poisson models: if the variance far exceeds the mean, standard errors are too small and you will declare spurious significance — switch to negative binomial or use a quasi-Poisson correction.

How this powers ML / AI

GLMs are the Rosetta Stone connecting statistics to deep learning's output layers and losses. A neural network's final layer plus its loss is *exactly* a GLM on learned features: a linear output with MSE loss is Gaussian/identity (regression); a sigmoid output with binary cross-entropy is Bernoulli/logit; a softmax output with categorical cross-entropy is the multinomial GLM; and predicting counts with a log-link exponential output and Poisson loss is Poisson regression. Choosing a network's output activation and loss function *is* choosing a GLM's link and distribution. This is why the right loss is not arbitrary — it is the negative log-likelihood of the response's distribution, exactly as in this chapter.

Try it in NumPy

```
import numpy as np, statsmodels.api as sm
rng = np.random.default_rng(0)
n = 300
x = rng.normal(size=n)
mu = np.exp(0.5 + 0.8*x)                # log link -> positive mean
y = rng.poisson(mu)                     # count response
X = sm.add_constant(x)
pois = sm.GLM(y, X, family=sm.families.Poisson()).fit()
```

```
print(pois.params.round(3))                          # ~[0.5, 0.8]
print("rate ratio per unit x:", np.exp(pois.params[1]).round(3))  # exp(beta)
```

Chapter summary

- A GLM has a **random component** (exponential-family distribution), a **linear predictor** $\boldsymbol{\beta}^\top \mathbf{x}$, and a **link function** g.
- The **link** generalizes regression: identity → linear, logit → logistic, log → Poisson.
- Counts use **Poisson** regression (e^{β_j} is a rate ratio); overdispersion calls for the negative binomial.
- The exponential-family structure gives one concave likelihood and one fitting algorithm (IRLS).
- A neural net's **output activation + loss** is exactly a GLM's link + distribution — which is why the loss equals the negative log-likelihood.

10 Nonlinear and Nonparametric Regression

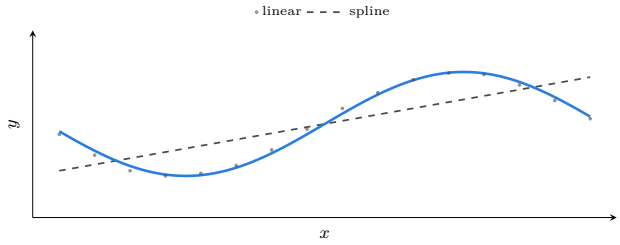
Level: Advanced

When the relationship is genuinely curved and you do not know its form, you need methods that let the *data* dictate the shape. This chapter surveys the flexible end of regression: polynomials and splines, local and kernel regression, generalized additive models, regression trees and forests, and Gaussian processes. These are the bridge from the rigid linear model to the universal function approximators of deep learning.

10.1 Polynomials and splines

Intuition

A high-degree polynomial can fit any wiggle, but it oscillates wildly between points and explodes at the boundaries (**Runge's phenomenon**). **Splines** fix this by stitching together low-degree (usually cubic) polynomial pieces at knots, forced to join smoothly. *Regression splines* place a handful of knots; *smoothing splines* put a knot at every point but add a roughness penalty $\lambda \int f''(x)^2\,dx$ — regularization (Chapter 7) reappearing to control wiggliness.



10.2 Local and kernel regression

Definition 10.1. Local regression (LOESS/LOWESS) estimates $\hat{f}(x_0)$ by fitting a low-degree polynomial to the points *near* x_0, weighted by distance through a **kernel**. **Kernel regression** (Nadaraya–Watson) is the simplest case: a locally weighted average,

$$\hat{f}(x_0) = \frac{\sum_i K_h(x_0 - x_i)\, y_i}{\sum_i K_h(x_0 - x_i)},$$

with bandwidth h setting the neighborhood size.

Intuition

The bandwidth h is the bias–variance dial in disguise: small h follows the data closely (low bias, high variance, jagged); large h smooths heavily (high bias, low variance, flat). These methods are **nonparametric** — they keep all the training data and have no fixed set of coefficients, so model complexity grows with the data rather than being fixed in advance.

10.3 Generalized additive models

Definition 10.2. A **GAM** keeps the additive, interpretable structure of linear regression but replaces each linear term with a smooth function:

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \varepsilon,$$

where each f_j is typically a spline. You see each predictor's individual nonlinear effect while avoiding the curse of dimensionality of a full p-dimensional smoother.

10.4 Trees, forests, and boosting

Intuition

A **regression tree** recursively splits the predictor space into boxes and predicts the mean response in each — a piecewise-constant surface, completely nonparametric and able to capture interactions automatically. A single tree is high-variance, so we average many: **random forests** (bagging decorrelated trees) and **gradient boosting** (adding trees that fit the residuals of the current ensemble) are among the most accurate off-the-shelf regressors for tabular data, routinely beating both linear models and neural networks there.

10.5 Gaussian process regression

Key idea

Gaussian process (GP) regression is the Bayesian nonparametric ideal: instead of a single fitted curve, it places a distribution over *all* smooth functions consistent with the data, yielding a predictive mean *and* calibrated uncertainty that widens away from observed points. A kernel encodes assumed smoothness; the posterior is computed in closed form via a Cholesky solve. GPs are the workhorse of **Bayesian optimization** (used to tune ML hyperparameters), where knowing *where the model is uncertain* is as valuable as the prediction itself.

Common pitfall

Flexibility is a double-edged sword. Nonparametric methods overfit eagerly — their smoothing parameters (h, knot count, tree depth, kernel scale) *must* be set by cross-validation, not eyeballed. They also suffer the **curse of dimensionality**: local methods and kernels degrade badly as p grows because "nearby" points become impossibly sparse. And kernel/GP methods cost $O(n^2)$–$O(n^3)$, limiting them to modest n without approximations. For high-dimensional, large-n problems, structured models (trees) or neural networks usually win.

How this powers ML / AI

This chapter is the conceptual on-ramp to deep learning. Neural networks are flexible nonparametric regressors — the **universal approximation theorem** says a sufficiently wide network can approximate any continuous function, just like splines or kernels, but they scale to high dimensions where classical nonparametrics fail. The connections are deep and literal: a wide random network converges to a Gaussian process (the Neural Tangent Kernel); the kernel trick that powers kernel regression also powers support vector regression; and gradient-boosted trees remain the state of the art for tabular regression, often outperforming deep nets. The trade-off you have seen — flexibility versus overfitting, controlled by a smoothing knob and cross-validation — is exactly how we reason about network capacity.

Try it in NumPy

```
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
rng = np.random.default_rng(0)
X = np.sort(rng.uniform(-3, 3, 60)).reshape(-1, 1)
y = np.sin(X).ravel() + rng.normal(0, 0.1, 60)
rf = RandomForestRegressor(n_estimators=200, random_state=0).fit(X, y
    )
gp = GaussianProcessRegressor(kernel=RBF(1.0), alpha=0.01).fit(X, y)
mean, std = gp.predict(X, return_std=True)   # GP gives mean AND
    uncertainty
print("RF R^2:", round(rf.score(X, y), 3), " GP mean std:", round(std
    .mean(), 3))
```

Chapter summary

- **Splines** stitch smooth polynomial pieces; smoothing splines add a roughness penalty (regularization again).
- **Local/kernel** regression averages nearby points; the bandwidth h is the bias–variance dial.
- **GAMs** keep additivity with smooth per-predictor effects; **trees/forests/boosting** are top tabular regressors.
- **Gaussian processes** give a posterior over functions with calibrated uncertainty — the engine of Bayesian optimization.
- All are flexible nonparametric regressors needing CV-tuned smoothing; neural nets extend this to high dimensions (universal approximation, NTK).

PART V

Advanced and Applications

11

Robust, Quantile, and Bayesian Regression

Level: Expert

Ordinary least squares is optimal under Gaussian noise — but real data has outliers, heavy tails, and asymmetric costs, and sometimes we want full probability distributions over our answers rather than point estimates. This chapter covers three expert extensions: **robust** regression (resistant to outliers), **quantile** regression (modeling the whole conditional distribution), and **Bayesian** regression (coherent uncertainty). Each changes the loss or the inference, not the underlying linear structure.

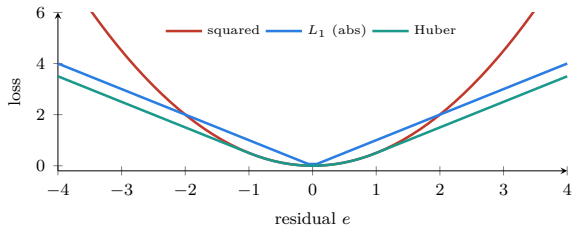
11.1 Robust regression

Intuition

Because least squares *squares* residuals, a single far-off point can dominate the fit — one outlier with residual 10 contributes as much as a hundred points with residual 1. Robust methods curb this by penalizing large residuals less aggressively.

Definition 11.1. Robust regression replaces the squared loss with a loss that grows slowly for large residuals:

- **Least absolute deviations** (L_1): minimize $\sum_i |e_i|$ — fits the conditional *median*, far less sensitive to outliers.
- **Huber loss**: quadratic for small residuals, linear beyond a threshold δ — a smooth compromise that is efficient *and* robust.



Intuition

The Huber curve is the practical sweet spot: near zero it behaves like least squares (statistically efficient when noise is Gaussian), but its linear tails stop outliers from hijacking the fit. This exact loss is widely used in machine learning — e.g. the `smooth_l1` / Huber loss in object detection and reinforcement learning — precisely for its robustness.

11.2 Quantile regression

Key idea

Ordinary regression models the conditional *mean*; **quantile regression** models any conditional *quantile* τ (e.g. the median $\tau = 0.5$, or the 90th percentile) by minimizing the asymmetric **pinball loss**

$$\rho_\tau(e) = \begin{cases} \tau\, e & e \geq 0 \\ (\tau - 1)\, e & e < 0 \end{cases}.$$

Fitting several τ's traces out the full conditional distribution, not just its center — revealing how spread and skew change with the predictors.

Intuition

Quantile regression answers questions the mean cannot: "what is a *pessimistic* (10th-percentile) delivery time?" or "how do the *top earners*, not the average, respond to education?" It needs no distributional assumption and is naturally robust. In ML it underlies **prediction intervals** and probabilistic forecasting — gradient-boosting and neural models routinely output multiple quantiles to quantify uncertainty.

11.3 Bayesian regression

Definition 11.2. Bayesian linear regression treats the coefficients as random, combining a **prior** $p(\boldsymbol{\beta})$ with the **likelihood** $p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\beta})$ to obtain a **posterior** via Bayes' theorem:

$$p(\boldsymbol{\beta} \mid \mathcal{D}) \;\propto\; p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\beta})\, p(\boldsymbol{\beta}).$$

Rather than one $\hat{\boldsymbol{\beta}}$, you get a whole distribution over plausible lines.

Key idea

With a Gaussian prior and Gaussian likelihood the posterior is Gaussian in closed form, and its mean is *exactly the ridge estimate* of Chapter 7 — regularization is a Gaussian prior, made explicit. The payoff is honest uncertainty: every prediction comes with a **posterior predictive** distribution whose width reflects both noise and parameter uncertainty, and which widens where data is scarce. Priors also stabilize estimation when data is limited or predictors collinear.

Common pitfall

Pitfalls across the three. **Robust:** most robust losses are non-quadratic, so there is no closed form and the optimization can have local issues; tune the threshold δ. **Quantile:** fitted quantile curves for different τ can *cross* (a 90th percentile below the 50th), which is logically impossible — use non-crossing constraints. **Bayesian:** results depend on the prior, and beyond the conjugate Gaussian case the posterior has no closed form, requiring MCMC or variational approximations that demand convergence checks. None of these is a free lunch — they buy robustness or uncertainty at the cost of computation or assumptions.

How this powers ML / AI

All three power modern AI. **Huber/smooth-L_1 loss** is standard in object detection (Faster R-CNN) and deep reinforcement learning (DQN) for stable training under outliers. **Quantile regression** is the backbone of probabilistic forecasting and of distributional RL (QR-DQN), where an agent learns the full distribution of returns, not just the mean. **Bayesian regression** is the conceptual seed of *Bayesian deep learning*: Bayesian neural networks, variational inference, Monte-Carlo dropout, and Laplace approximations all aim to put a posterior over network weights so the model knows what it does not know — essential for safety-critical AI, active learning, and out-of-distribution detection.

Try it in NumPy

```
import numpy as np
from sklearn.linear_model import HuberRegressor, QuantileRegressor,
    BayesianRidge
rng = np.random.default_rng(0)
X = rng.normal(size=(100, 1)); y = 2*X.ravel() + rng.normal(0, 0.5,
    100)
y[:5] += 15                                        # inject outliers
print("OLS-ish Huber slope:", HuberRegressor().fit(X, y).coef_[0].
    round(3))
q90 = QuantileRegressor(quantile=0.9, alpha=0).fit(X, y)   # 90th
    percentile fit
bayes = BayesianRidge().fit(X, y)
mean, std = bayes.predict(X[:3], return_std=True)          #
    prediction + uncertainty
print("Bayesian pred std:", std.round(3))
```

Chapter summary

- **Robust** regression (L_1, Huber) resists outliers by penalizing large residuals less than squared loss; Huber is the efficient–robust compromise.
- **Quantile** regression models conditional quantiles via the asymmetric pinball loss, giving the whole distribution and prediction intervals.
- **Bayesian** regression returns a posterior over coefficients; with Gaussian prior/likelihood its mean is the ridge estimate, plus calibrated predictive uncertainty.
- Each trades computation or assumptions for robustness or uncertainty — no free lunch.
- These power Huber loss (detection/RL), distributional RL and probabilistic forecasting, and Bayesian deep learning for "knowing what you don't know."

12

Evaluation, Pitfalls, and Causality

Level: Expert

A regression can be technically correct and practically misleading. This chapter is the practitioner's survival guide: how to measure performance with the right metric, the classic traps that fool even experts, and the bright line between *prediction* and *causation* — the distinction that determines whether a coefficient can guide a decision or merely describe an association.

12.1 Regression metrics

Definition 12.1. Common metrics for a continuous response:

- **MSE / RMSE**: $\frac{1}{n}\sum e_i^2$ and its square root; RMSE is in the response's units and penalizes large errors heavily.
- **MAE**: $\frac{1}{n}\sum |e_i|$; robust to outliers, equal weight to all errors.
- R^2: fraction of variance explained (Chapter 2); good for comparison, poor as an absolute quality measure.
- **MAPE**: mean absolute *percentage* error; scale-free but undefined/unstable near zero.

Intuition

The choice encodes your loss. If a \$10k error is twice as bad as two \$5k errors, use RMSE; if all errors hurt proportionally, use MAE. Reporting RMSE when the business cares about MAE (or vice versa) optimizes the wrong thing. Always compare against a **baseline** — predicting the mean, or last year's value — so a number like "RMSE = 4.2" has meaning.

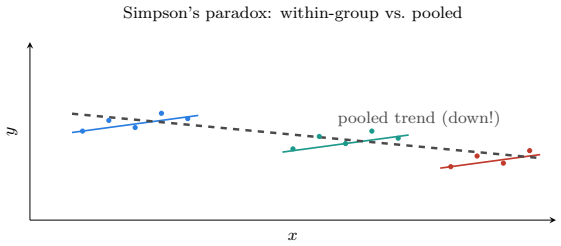
12.2 Classic pitfalls

Common pitfall

The hall of fame of regression mistakes:

- **Extrapolation:** predictions outside the training range of **x** are unreliable — the fitted form may not hold there.
- **Data leakage:** any information from the test set (or the future) entering training inflates measured performance and collapses in production — the #1 cause of "great in the notebook, terrible in production."
- **Simpson's paradox:** a trend within every subgroup can reverse when groups are pooled; an omitted grouping variable flips the sign.
- **Multiple comparisons / p-hacking:** test enough predictors and some look "significant" by chance.
- **Outliers and high-leverage points:** a few extreme points can dominate the fit

(Chapter 11); always inspect them.
- **Overfitting to the validation set:** tuning hundreds of configurations against one validation split silently overfits it — use nested CV.



12.3 Correlation is not causation

Key idea

A regression coefficient measures *association* in the observed data, not the effect of *intervening*. Ice-cream sales predict drownings (both driven by summer heat — a **confounder**), but banning ice cream saves no one. To read a coefficient causally you need either a randomized experiment or strong, explicit assumptions (no unmeasured confounders) plus the right adjustment set. Adding controls can *help* (block confounders) or *hurt* (conditioning on a collider or mediator induces spurious associations) — which variables to include is a causal question a regression alone cannot answer.

Intuition

A useful mantra: *regression adjusts for what you put in it, and nothing else.* It cannot adjust for variables you did not measure, and it will happily hand you a precise, significant, and completely non-causal coefficient. Modern causal inference (potential outcomes, DAGs, instrumental variables, difference-in-differences, regression discontinuity) builds *on top of* regression but adds the design and assumptions that license a causal reading.

12.4 A practical workflow

1. Plot the data and define the goal (predict vs. explain) and the loss/metric.
2. Split off a test set *first*; do all exploration and tuning on the rest.
3. Build features in a leak-free pipeline; start simple (linear) as a baseline.
4. Diagnose residuals; add complexity or regularization guided by cross-validation.
5. Report the right metric with uncertainty, against a baseline, on the untouched test set.
6. State assumptions explicitly — especially before any causal claim.

How this powers ML / AI

These lessons scale directly to large ML and AI systems. **Data leakage** (e.g. train/test contamination, or future information in features) is the most common reason a model that dazzles offline fails in deployment, and it is rampant in large pipelines. **Distribution shift** is extrapolation at scale — models degrade when production data drifts from training data. **Spurious correlations / shortcut learning** is Simpson's paradox in disguise: deep nets latch onto confounded cues (backgrounds, watermarks) that do not generalize. And **causal ML** — uplift modeling, off-policy evaluation in RL, treatment-effect estimation — exists

precisely because predictive accuracy does not imply you can act on a feature. Regression's hard-won cautions are the foundation of trustworthy AI.

Try it in NumPy

```
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error,
    r2_score
rng = np.random.default_rng(0)
y = rng.normal(50, 10, 200)
yhat = y + rng.normal(0, 4, 200); yhat[:3] += 40        # a few big
    misses
print("RMSE:", round(mean_squared_error(y, yhat, squared=False), 2))
     # outlier-sensitive
print("MAE :", round(mean_absolute_error(y, yhat), 2))
     # outlier-robust
print("R^2 :", round(r2_score(y, yhat), 3))
# Always compare to a baseline (predict the mean):
print("baseline RMSE:", round(mean_squared_error(y, np.full_like(y, y
    .mean()),
                                                 squared=False), 2))
```

Chapter summary

- Pick the metric that matches your cost: **RMSE** (penalizes big errors), **MAE** (robust), R^2 (relative); always compare to a baseline.
- Beware **extrapolation**, **data leakage**, **Simpson's paradox**, multiple comparisons, outliers, and validation-set overfitting.
- A coefficient is **association, not causation**; causal reading needs experiments or explicit assumptions and the right adjustment set.
- Regression adjusts only for variables you include — never for unmeasured confounders.
- These pitfalls scale to ML as leakage, distribution shift, shortcut learning, and the motivation for causal ML.

13

Regression in Machine Learning, Deep Learning, and AI

Level: Capstone

This capstone ties the book together by arguing a single thesis: *regression is not just one tool in machine learning — it is the conceptual skeleton of the entire field.* Supervised learning is regression and classification at scale. A neural network is a flexible regression function. Its output layer is a linear, logistic, or softmax regression. Its loss is a regression or classification likelihood. Its training is least-squares-style optimization. Even reinforcement learning and generative AI reduce, at their core, to regression problems. We revisit each idea from the book and show where it lives in modern AI.

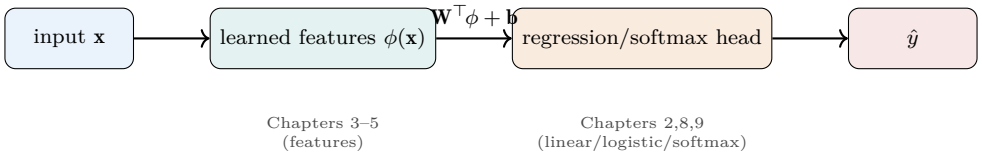
13.1 The neural network as flexible regression

Key idea

A feed-forward neural network computes

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}^{(L)}\,\phi\big(\cdots\phi(\mathbf{W}^{(1)}\mathbf{x}+\mathbf{b}^{(1)})\cdots\big)+\mathbf{b}^{(L)},$$

which is exactly the basis-function regression of Chapter 5 — except the basis $\phi(\cdot)$ is *learned* rather than hand-designed. Each layer is the matrix multiply $\mathbf{XW}$ of Chapter 3; the final layer is a multiple/logistic/softmax regression on the learned representation. The **universal approximation theorem** guarantees this flexible regressor can approximate any continuous function. Deep learning is feature learning plus regression.



13.2 Loss functions are negative log-likelihoods

Key idea

The loss a network minimizes is precisely the GLM correspondence of Chapter 9: the output activation and loss together specify the response's distribution, and the loss is its negative log-likelihood.

Task	Output activation	Loss	Regression analog
Continuous value	identity (linear)	MSE / L_2	linear regression (Ch. 2)
Robust regression	identity	Huber / smooth-L_1	robust regression (Ch. 11)
Binary label	sigmoid	binary cross-entropy	logistic regression (Ch. 8)
Multiclass label	softmax	categorical cross-entropy	softmax regression (Ch. 8)
Counts	exp (log link)	Poisson loss	Poisson GLM (Ch. 9)
Quantiles / intervals	identity	pinball loss	quantile regression (Ch. 11)

Choosing a network's head and loss *is* choosing a GLM. The right loss is never arbitrary — it is the likelihood of the data.

13.3 Optimization: least squares by gradient descent

Intuition

Linear and logistic regression have closed forms or quickly-converging solvers; deep networks do not, so they minimize the *same* losses by **gradient descent** and its stochastic variants (SGD, Adam, AdamW). The objective is identical in spirit to minimizing RSS — only the parameter count and nonconvexity change. **Backpropagation** is just the chain rule computing the gradient of the regression loss with respect to every weight. The intuition you built fitting a line carries all the way to training billion-parameter models.

13.4 Regularization, revisited at scale

- **Weight decay** = ridge (L_2) penalty (Chapter 7); the default in AdamW for training Transformers.
- L_1 / **sparsity** penalties prune and compress networks.
- **Early stopping** and **dropout** are implicit regularizers managing the same bias–variance tradeoff (Chapter 6).
- **Cross-validation** remains the universal protocol for tuning every hyperparameter.

13.5 Regression at the frontier of AI

How this powers ML / AI

The regression view illuminates even the most modern systems:

- **Large language models** predict the next token by **softmax regression** over the vocabulary on top of a Transformer's learned representation — cross-entropy, exactly as in Chapter 8. Adapting an LLM to output numbers uses lightweight **regression heads** (a linear layer, or iterative-refinement heads like RELISH) trained with MSE.
- **Diffusion models** generate images by training a network to **regress** the noise added to data (an MSE objective) — generation as denoising regression.
- **Reinforcement learning** learns value functions by regressing returns (TD/Monte-Carlo targets); **distributional RL** uses quantile regression (Chapter 11) to predict the whole return distribution.
- **Scaling laws** are themselves power-law *regressions* of loss against model size, data, and

compute — regression used to forecast the behavior of AI systems.
- **Uncertainty & safety**: Bayesian regression (Chapter 11) seeds Bayesian neural networks, MC-dropout, and Laplace approximations so models know what they don't know.

13.6 The thread, end to end

Key idea

Trace one idea through the whole book and into AI: *fit a function to data by minimizing a loss, and regularize so it generalizes.* Chapter 2 fit a line by least squares; Chapter 3 made it multivariate and geometric; Chapter 8 swapped the loss for cross-entropy; Chapter 9 unified losses as likelihoods; Chapter 7 added penalties; Chapter 6 explained generalization; Chapter 10 made the function flexible. A deep network is all of these at once, learned end to end. **Regression is where machine learning begins, and it never really leaves.**

Try it in NumPy

```
import torch, torch.nn as nn
# A 1-layer net with identity output + MSE == linear regression by gradient descent
X = torch.randn(200, 3); beta = torch.tensor([2.0, -1.0, 0.5])
y = X @ beta + 0.1*torch.randn(200)
net = nn.Linear(3, 1)
opt = torch.optim.AdamW(net.parameters(), lr=0.05, weight_decay=1e-3)  # = ridge
lossf = nn.MSELoss()
for _ in range(400):
    opt.zero_grad()
    loss = lossf(net(X).squeeze(), y)     # minimize RSS by gradient descent
    loss.backward(); opt.step()
print(net.weight.data.round(decimals=3))  # recovers ~[2.0, -1.0, 0.5]
```

Chapter summary

- A neural network is **flexible regression**: learned features $\phi(\mathbf{x})$ plus a linear/logistic/softmax **head** (Chapters 3, 8).
- Its **loss** is a negative log-likelihood — output activation + loss = a GLM (Chapter 9); the right loss is the data's likelihood.
- Training minimizes that loss by **gradient descent** (backprop = chain rule), generalizing least squares.
- **Weight decay** is ridge; dropout, early stopping, and cross-validation manage the bias–variance tradeoff at scale.
- LLMs (softmax/regression heads), diffusion (noise regression), RL (value/quantile regression), and scaling laws are all regression — the idea that starts the book and underpins modern AI.

A

Formulas and Method Reference

A compact reference to the book's key formulas and a decision guide for choosing a method.

A.1 Core formulas

Quantity	**Formula**
Simple slope / intercept	$\hat{\beta}_1 = \dfrac{\mathrm{Cov}(x,y)}{\mathrm{Var}(x)}, \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$
Residual sum of squares	$\mathrm{RSS} = \sum_i (y_i - \hat{y}_i)^2$
Coefficient of determination	$R^2 = 1 - \mathrm{RSS}/\mathrm{TSS}, \quad \mathrm{TSS} = \sum_i (y_i - \bar{y})^2$
Adjusted R^2	$1 - (1 - R^2)\dfrac{n-1}{n-p-1}$
Normal equations (OLS)	$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$
Hat (projection) matrix	$\mathbf{H} = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top, \quad \hat{\mathbf{y}} = \mathbf{H}\mathbf{y}$
Coefficient covariance	$\widehat{\mathrm{Cov}}(\hat{\boldsymbol{\beta}}) = \hat{\sigma}^2 (\mathbf{X}^\top \mathbf{X})^{-1}$
Ridge estimate	$\hat{\boldsymbol{\beta}}_{\text{ridge}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$
Lasso objective	$\min_{\boldsymbol{\beta}} \ \lVert \mathbf{y} - \mathbf{X}\boldsymbol{\beta} \rVert^2 + \lambda \sum_j \lvert \beta_j \rvert$
Elastic net penalty	$\lambda \left[\alpha \sum_j \lvert \beta_j \rvert + (1-\alpha) \sum_j \beta_j^2 \right]$
Logistic model	$p(\mathbf{x}) = \sigma(\boldsymbol{\beta}^\top \mathbf{x}), \quad \sigma(z) = 1/(1+e^{-z})$
Binary cross-entropy	$-\sum_i [y_i \ln p_i + (1-y_i) \ln(1-p_i)]$
Softmax	$\Pr(y = k \mid \mathbf{x}) = e^{\boldsymbol{\beta}_k^\top \mathbf{x}} / \sum_j e^{\boldsymbol{\beta}_j^\top \mathbf{x}}$
GLM link	$g(\mu) = \boldsymbol{\beta}^\top \mathbf{x}, \quad \mu = g^{-1}(\boldsymbol{\beta}^\top \mathbf{x})$
VIF	$\mathrm{VIF}_j = 1/(1 - R_j^2)$
Bias–variance	$\mathbb{E}[(y - \hat{f})^2] = \mathrm{Bias}^2 + \mathrm{Var} + \sigma^2$
AIC / BIC	$-2 \ln \hat{L} + 2k \ \ / \ \ -2 \ln \hat{L} + k \ln n$
Huber loss	quadratic for $\lvert e \rvert \le \delta$, linear beyond
Pinball (quantile) loss	$\rho_\tau(e) = \max(\tau e, (\tau - 1)e)$

A.2 Which method should I use?

Situation	Recommended method
Continuous y, few predictors	ordinary least squares (Ch. 2, 3)
Many / correlated predictors	ridge; lasso/elastic net for selection (Ch. 7)
$p > n$ (wide data)	ridge or elastic net (Ch. 7)
Binary outcome	logistic regression (Ch. 8)
Multiclass outcome	softmax / multinomial logistic (Ch. 8)
Count outcome	Poisson / negative-binomial GLM (Ch. 9)
Positive, skewed outcome	Gamma GLM or log-transform (Ch. 9)
Curved relationship, known form	polynomial / basis features (Ch. 5)
Curved, unknown form, low-dim	splines, GAMs, LOESS (Ch. 10)
Tabular, high accuracy	gradient boosting / random forest (Ch. 10)
Need uncertainty / small data	Gaussian process, Bayesian regression (Ch. 10, 11)
Outliers present	Huber / least-absolute-deviations (Ch. 11)
Need intervals / tails	quantile regression (Ch. 11)
High-dim, large n, complex	neural network (Ch. 13)

A.3 Modeling checklist

1. Define the goal: prediction or inference? Choose the loss/metric accordingly.
2. Split off a test set *before* any exploration; build features in a leak-free pipeline.
3. Fit a simple linear baseline; standardize predictors before regularizing.
4. Check residual plots (linearity, equal variance, normality, outliers, leverage).
5. Tune complexity / λ by cross-validation; prefer the simplest model within one SE of the best.
6. Report the chosen metric with uncertainty, against a baseline, on the untouched test set.
7. State assumptions explicitly; do not read coefficients causally without a design that licenses it.

A.4 Symbols

Symbol	Meaning
$y,\ \hat{y}$	response; fitted/predicted value
$\mathbf{x},\ \mathbf{X}$	predictor vector; design matrix ($n \times (p+1)$)
$\boldsymbol{\beta},\ \hat{\boldsymbol{\beta}}$	coefficient vector; its estimate
$\varepsilon,\ e$	error term; residual $y - \hat{y}$
λ	regularization strength
σ^2	irreducible noise variance
$\sigma(\cdot)$	logistic (sigmoid) function
$n,\ p$	number of observations; number of predictors