Classical Data Science

A Solutions Architect's Field Guide, Volume 1

Andrew R. Kane

Classical Data Science
A Solutions Architect's Field Guide, Volume 1

Copyright © 2026 Andrew R. Kane.

This book is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). You may copy, share and adapt it for any purpose, including commercial use. You must credit the author, and share any changed version under the same license. License text: https://creativecommons.org/licenses/by-sa/4.0/

The companion notebooks are at https://github.com/arkane-dev/book-companion-notebooks and are licensed under the MIT License. This edition matches the release tagged `book1-v1.1`.

ISBN: 9798142514622 (paperback)

First edition, 2026.

The code and methods in this book are for teaching. They come with no warranty. Test them before you rely on them.

Amazon Web Services, AWS and Amazon SageMaker are trademarks of Amazon.com, Inc. or its affiliates.

andrewrkane.com

Table of contents

Preface **i**
How to Read This Book . . . i
Notation and Conventions . . . ii
Prerequisites Map . . . ii

1 History & Landscape **1**
1.1 Motivation . . . 1
1.2 Exposition . . . 1
1.2.1 The Three Eras . . . 1
1.2.2 The Landscape Today . . . 3
1.3 Trade-offs and Limits . . . 4
1.4 Procedures . . . 5
1.5 Worked Example . . . 8
1.6 History and Further Reading . . . 10
1.7 How This Connects . . . 10
1.8 Exercises . . . 10
1.9 Mental Model . . . 11

2 Business Problem Framing **13**
2.1 Motivation . . . 13
2.2 Exposition . . . 13
2.2.1 Understanding the Business Goal . . . 13
2.2.2 Defining the ML Problem . . . 15
2.2.3 Defining Success . . . 15
2.2.4 The Precision-Recall Trade-off as a Business Decision . . . 16
2.2.5 The ML Problem Type Taxonomy . . . 17
2.3 Trade-offs and Limits . . . 17
2.4 Procedures . . . 18
2.5 Worked Example . . . 20
2.6 History and Further Reading . . . 22
2.7 How This Connects . . . 22
2.8 Exercises . . . 22
2.9 Mental Model . . . 23

3 Maths Foundations: Statistics, Probability & Linear Algebra 25
3.1 Motivation . . . 25
3.2 Exposition . . . 26
3.2.1 Probability . . . 26
3.2.2 Common Probability Distributions . . . 27
3.2.3 Statistics . . . 29
3.2.4 Linear Algebra . . . 30
3.2.5 Linear Regression as a Matrix Equation . . . 33
3.3 Trade-offs and Limits . . . 33
3.4 Procedures . . . 34
3.5 Worked Example . . . 37
3.6 AWS SageMaker Notes . . . 39
3.7 History and Further Reading . . . 40
3.8 How This Connects . . . 40
3.9 Exercises . . . 41
3.10 Mental Model . . . 42

4 Maths Foundations: Calculus & Optimisation 43
4.1 Motivation . . . 43
4.2 Exposition . . . 43
4.2.1 Derivatives and Gradients . . . 43
4.2.2 Gradient Descent . . . 44
4.2.3 The Learning Rate . . . 45
4.2.4 The Chain Rule and Backpropagation . . . 45
4.2.5 Loss Functions . . . 46
4.2.6 Variants of Gradient Descent . . . 47
4.2.7 Momentum and Adam . . . 47
4.2.8 When scipy Can Help, and When It Can't . . . 48
4.3 Trade-offs and Limits . . . 48
4.4 Procedures . . . 49
4.5 Worked Example . . . 53
4.6 AWS SageMaker Notes . . . 56
4.7 History and Further Reading . . . 56
4.8 How This Connects . . . 57
4.9 Exercises . . . 57
4.10 Mental Model . . . 58

5 Information Theory Basics 59
5.1 Motivation . . . 59
5.2 Exposition . . . 60
5.2.1 Entropy: How Uncertain Is a Distribution? . . . 60
5.2.2 Cross-Entropy: The Cost of Being Wrong . . . 60
5.2.3 KL Divergence: How Far Is Q From P? . . . 61

5.2.4 Mutual Information: How Much Does Knowing X Tell You About Y? 62
5.2.5 How the Four Fit Together . . . 63
5.3 Trade-offs and Limits . . . 63
5.4 Procedures . . . 64
5.5 Worked Example . . . 67
5.6 History and Further Reading . . . 70
5.7 How This Connects . . . 70
5.8 Exercises . . . 70
5.9 Mental Model . . . 71

6 Python Data Science Ecosystem 73
6.1 Motivation . . . 73
6.2 Exposition . . . 73
6.2.1 The Layered Stack . . . 73
6.2.2 NumPy: Fast Arrays . . . 74
6.2.3 pandas: Labelled Tabular Data . . . 75
6.2.4 matplotlib: The Low-Level Plotting Layer . . . 75
6.2.5 seaborn: Statistical Visualisation . . . 76
6.2.6 scipy: Scientific Computing . . . 76
6.2.7 scikit-learn: The ML Algorithm Layer . . . 76
6.2.8 Jupyter Notebooks: Literate Computing . . . 76
6.3 Trade-offs and Limits . . . 77
6.4 Procedures . . . 77
6.5 Worked Example . . . 80
6.6 History and Further Reading . . . 82
6.7 How This Connects . . . 83
6.8 Exercises . . . 83
6.9 Mental Model . . . 84

7 Data Cleaning & Wrangling 85
7.1 Motivation . . . 85
7.2 Exposition . . . 87
7.2.1 Missing Values . . . 87
7.2.2 Outliers . . . 87
7.2.3 Duplicates . . . 89
7.2.4 Type Coercion . . . 89
7.2.5 Encoding Errors . . . 89
7.2.6 Inconsistent Categories . . . 90
7.2.7 Schema Violations . . . 90
7.3 Trade-offs and Limits . . . 91
7.4 Procedures . . . 91
7.5 Worked Example . . . 95
7.6 History and Further Reading . . . 98

7.7 How This Connects . . . 98
7.8 Exercises . . . 98
7.9 Mental Model . . . 99

8 SQL for Data Science 101
8.1 Motivation . . . 101
8.2 Exposition . . . 101
8.2.1 GROUP BY and Aggregations . . . 101
8.2.2 CTEs — Common Table Expressions . . . 102
8.2.3 Window Functions — The Most Powerful Analytical Tool . . . 103
8.2.4 Joins for Analysis . . . 104
8.2.5 Using SQL at Scale . . . 104
8.3 Trade-offs and Limits . . . 105
8.4 Procedures . . . 106
8.5 Worked Example . . . 110
8.6 History and Further Reading . . . 114
8.7 How This Connects . . . 114
8.8 Exercises . . . 114
8.9 Mental Model . . . 115

9 Exploratory Data Analysis & Visualisation 117
9.1 Motivation . . . 117
9.2 Exposition . . . 118
9.2.1 Univariate Analysis . . . 118
9.2.2 Bivariate Analysis . . . 118
9.2.3 Multivariate Analysis . . . 118
9.2.4 The EDA Mindset . . . 119
9.3 Trade-offs and Limits . . . 119
9.4 Procedures . . . 121
9.5 Worked Example . . . 125
9.6 History and Further Reading . . . 127
9.7 How This Connects . . . 127
9.8 Exercises . . . 128
9.9 Mental Model . . . 128

10 The Data Science Lifecycle 129
10.1 Motivation . . . 129
10.2 Exposition . . . 130
10.2.1 The Stages, Mapped to This Book . . . 130
10.2.2 Why the Textbook Diagram Lies . . . 131
10.2.3 Why Projects Die at Each Stage . . . 131
10.3 Trade-offs and Limits . . . 132
10.4 Procedures . . . 133

10.5 Worked Example . . . 137
10.6 History and Further Reading . . . 139
10.7 How This Connects . . . 140
10.8 Exercises . . . 140
10.9 Mental Model . . . 141

11 Supervised Learning: Overview of Approaches 143
11.1 Motivation . . . 143
11.2 Exposition . . . 144
11.2.1 The Bias-Variance Trade-Off . . . 144
11.2.2 Linear Models . . . 144
11.2.3 Support Vector Machines . . . 146
11.2.4 Tree-Based Models . . . 147
11.2.5 Distance-Based Models . . . 147
11.2.6 Probabilistic Models . . . 147
11.2.7 The No Free Lunch Theorem . . . 148
11.3 Trade-offs and Limits . . . 148
11.4 Procedures . . . 149
11.5 Worked Example . . . 151
11.6 History and Further Reading . . . 153
11.7 How This Connects . . . 153
11.8 Exercises . . . 153
11.9 Mental Model . . . 154

12 Ensemble Methods 155
12.1 Motivation . . . 155
12.2 Exposition . . . 155
12.2.1 Bagging — Reducing Variance . . . 155
12.2.2 Random Forests — Bagging Plus Feature Randomness . . . 156
12.2.3 Boosting — Reducing Bias . . . 156
12.2.4 Gradient Boosting — Boosting as Gradient Descent . . . 156
12.2.5 XGBoost and LightGBM . . . 157
12.2.6 Stacking — Learning How to Combine . . . 157
12.3 Trade-offs and Limits . . . 158
12.4 Procedures . . . 158
12.5 Worked Example . . . 161
12.6 History and Further Reading . . . 163
12.7 How This Connects . . . 163
12.8 Exercises . . . 164
12.9 Mental Model . . . 164

13 Cross-Validation & Model Selection 165
13.1 Motivation . . . 165

13.2 Exposition . . . 165
13.2.1 The Problem With a Single Train/Test Split . . . 165
13.2.2 k-Fold Cross-Validation . . . 165
13.2.3 Stratified k-Fold . . . 166
13.2.4 Time-Series Cross-Validation . . . 166
13.2.5 The Overfitting-to-Test-Set Problem . . . 166
13.2.6 Nested Cross-Validation . . . 167
13.3 Trade-offs and Limits . . . 167
13.4 Procedures . . . 167
13.5 Worked Example . . . 170
13.6 History and Further Reading . . . 172
13.7 How This Connects . . . 172
13.8 Exercises . . . 172
13.9 Mental Model . . . 173

14 Feature Engineering **175**
14.1 Motivation . . . 175
14.2 Exposition . . . 175
14.2.1 Numeric Feature Transformations . . . 175
14.2.2 Categorical Feature Encoding . . . 178
14.2.3 Datetime Feature Extraction . . . 178
14.2.4 Interaction Features . . . 180
14.2.5 Domain-Specific Features . . . 180
14.3 Trade-offs and Limits . . . 181
14.4 Procedures . . . 181
14.5 Worked Example . . . 184
14.6 History and Further Reading . . . 185
14.7 How This Connects . . . 185
14.8 Exercises . . . 186
14.9 Mental Model . . . 186

15 Feature Selection & Dimensionality Reduction **187**
15.1 Motivation . . . 187
15.2 Exposition . . . 187
15.2.1 The Curse of Dimensionality . . . 187
15.2.2 Feature Selection: Filter, Wrapper, Embedded . . . 188
15.2.3 Feature Extraction: PCA . . . 189
15.2.4 A Note on Kernel PCA . . . 190
15.2.5 t-SNE and UMAP: Visualisation, Not Modelling Features . . . 190
15.3 Trade-offs and Limits . . . 193
15.4 Procedures . . . 193
15.5 Worked Example . . . 197
15.6 AWS SageMaker Notes . . . 199

15.7 History and Further Reading . . . 200
15.8 How This Connects . . . 200
15.9 Exercises . . . 200
15.10 Mental Model . . . 201

16 Model Evaluation **203**
16.1 Motivation . . . 203
16.2 Exposition . . . 203
16.2.1 The Confusion Matrix . . . 203
16.2.2 Classification Metrics . . . 204
16.2.3 The Precision-Recall Trade-off . . . 204
16.2.4 Regression Metrics . . . 204
16.2.5 Calibration . . . 205
16.2.6 The Business Metric Gap . . . 205
16.3 Trade-offs and Limits . . . 205
16.4 Procedures . . . 205
16.5 Worked Example . . . 208
16.6 History and Further Reading . . . 210
16.7 How This Connects . . . 210
16.8 Exercises . . . 210
16.9 Mental Model . . . 211

17 Hyperparameter Optimisation **213**
17.1 Motivation . . . 213
17.2 Exposition . . . 213
17.2.1 Parameters vs. Hyperparameters . . . 213
17.2.2 Search Strategies . . . 214
17.2.3 The Validation Set Is Sacred . . . 214
17.3 Trade-offs and Limits . . . 215
17.4 Procedures . . . 215
17.5 Worked Example . . . 217
17.6 History and Further Reading . . . 219
17.7 How This Connects . . . 220
17.8 Exercises . . . 220
17.9 Mental Model . . . 221

18 Regularisation **223**
18.1 Motivation . . . 223
18.2 Exposition . . . 223
18.2.1 The Bias-Variance Trade-off . . . 223
18.2.2 Diagnosing with Learning Curves . . . 223
18.2.3 L2 Regularisation (Ridge) . . . 225
18.2.4 L1 Regularisation (Lasso) . . . 225

18.2.5 Elastic Net . . . 226
18.2.6 Dropout (Neural Networks) . . . 226
18.2.7 Early Stopping . . . 226
18.3 Trade-offs and Limits . . . 226
18.4 Procedures . . . 227
18.5 Worked Example . . . 229
18.6 History and Further Reading . . . 230
18.7 How This Connects . . . 230
18.8 Exercises . . . 231
18.9 Mental Model . . . 231

19 Unsupervised Learning **233**
19.1 Motivation . . . 233
19.2 Exposition . . . 233
19.2.1 K-Means Clustering . . . 233
19.2.2 Gaussian Mixture Models . . . 234
19.2.3 DBSCAN . . . 234
19.2.4 Hierarchical Clustering . . . 235
19.2.5 Isolation Forest . . . 235
19.3 Trade-offs and Limits . . . 235
19.4 Procedures . . . 236
19.5 Worked Example . . . 238
19.6 History and Further Reading . . . 239
19.7 How This Connects . . . 240
19.8 Exercises . . . 240
19.9 Mental Model . . . 241

20 Time Series Analysis **243**
20.1 Motivation . . . 243
20.2 Exposition . . . 243
20.2.1 Stationarity . . . 243
20.2.2 Autocorrelation and Partial Autocorrelation . . . 244
20.2.3 ARIMA: AR, I, MA . . . 244
20.2.4 Exponential Smoothing . . . 244
20.2.5 Beyond Classical Statistics: ML and Deep Learning . . . 244
20.3 Trade-offs and Limits . . . 245
20.4 Procedures . . . 245
20.5 Worked Example . . . 247
20.6 History and Further Reading . . . 249
20.7 How This Connects . . . 249
20.8 Exercises . . . 249
20.9 Mental Model . . . 250

Quick Reference **251**
Chapter 1 — History & Landscape . . . 251
Chapter 2 — Business Problem Framing . . . 254
Chapter 3 — Maths Foundations: Statistics, Probability & Linear Algebra . . . 256
Chapter 4 — Maths Foundations: Calculus & Optimisation . . . 261
Chapter 5 — Information Theory Basics . . . 266
Chapter 6 — Python Data Science Ecosystem . . . 270
Chapter 7 — Data Cleaning & Wrangling . . . 274
Chapter 8 — SQL for Data Science . . . 278
Chapter 9 — Exploratory Data Analysis & Visualisation . . . 284
Chapter 10 — The Data Science Lifecycle . . . 289
Chapter 11 — Supervised Learning: Overview of Approaches . . . 294
Chapter 12 — Ensemble Methods . . . 299
Chapter 13 — Cross-Validation & Model Selection . . . 301
Chapter 14 — Feature Engineering . . . 303
Chapter 15 — Feature Selection & Dimensionality Reduction . . . 306
Chapter 16 — Model Evaluation . . . 310
Chapter 17 — Hyperparameter Optimisation . . . 313
Chapter 18 — Regularisation . . . 315
Chapter 19 — Unsupervised Learning . . . 318
Chapter 20 — Time Series Analysis . . . 320

Answers to Exercises **325**
Chapter 1 — History & Landscape . . . 325
Chapter 2 — Business Problem Framing . . . 326
Chapter 3 — Maths Foundations: Statistics, Probability & Linear Algebra . . . 327
Chapter 4 — Maths Foundations: Calculus & Optimisation . . . 329
Chapter 5 — Information Theory Basics . . . 331
Chapter 6 — Python Data Science Ecosystem . . . 333
Chapter 7 — Data Cleaning & Wrangling . . . 334
Chapter 8 — SQL for Data Science . . . 336
Chapter 9 — Exploratory Data Analysis & Visualisation . . . 338
Chapter 10 — The Data Science Lifecycle . . . 339
Chapter 11 — Supervised Learning: Overview of Approaches . . . 342
Chapter 12 — Ensemble Methods . . . 344
Chapter 13 — Cross-Validation & Model Selection . . . 345
Chapter 14 — Feature Engineering . . . 347
Chapter 15 — Feature Selection & Dimensionality Reduction . . . 348
Chapter 16 — Model Evaluation . . . 349
Chapter 17 — Hyperparameter Optimisation . . . 350
Chapter 18 — Regularisation . . . 352
Chapter 19 — Unsupervised Learning . . . 353
Chapter 20 — Time Series Analysis . . . 354

References **357**

Preface

Solutions Architects sit at one of the most interesting intersections in modern technology: between what customers need and what's technically possible. For years, that meant knowing cloud infrastructure, distributed systems, and integration patterns. That's no longer enough. Customers now build machine learning pipelines and ask hard questions that can't be answered from a whitepaper.

Classical Data Science is Volume 1 of a series written for that reader: someone who already thinks in systems and already knows Python, but needs a rigorous, end-to-end path through data science before tackling modern AI in Volume 2. This book covers the foundations, data work, classical machine learning, evaluation, and project lifecycle that most practising data scientists spend most of their time in. It's less glamorous than large language models, but it's where most real problems still get solved.

The AWS Solutions Architect angle is not incidental. Every chapter that has a natural cloud-native counterpart includes an **AWS SageMaker Notes** sidebar, so the path from "I understand this technique" to "I can deploy this technique on AWS" stays short throughout.

How to Read This Book

Each chapter follows a consistent anatomy, adapted from the Diataxis documentation framework (Procida 2020):

Section	Purpose
Motivation	The problem that existed before this technique, in plain language
Exposition	The mechanism — why the technique works, not just how to invoke it
Trade-offs and Limits	Where the technique holds up, and where it quietly fails
Procedures	Formally labelled, task-oriented recipes with code listings

Section	Purpose
Worked Example	A narrated run showing the technique applied to a realistic scenario
AWS SageMaker Notes	The managed-service equivalent, and when it's worth the overhead
History and Further Reading	Context, and where to go deeper
How This Connects	Explicit links to chapters this one builds on and enables
Exercises	Difficulty-tagged practice, answers in Appendix B
Mental Model	A one-line distillation to carry forward

You don't have to read a chapter start to finish in one sitting. Many readers read Motivation through Trade-offs first, work through the Worked Example at a computer later, and keep Appendix A open as a desk reference while coding. Quizzes: andrewrkane.com/quiz/vol1.

Notation and Conventions

Formulas are typeset in standard mathematical notation (e.g. $\hat{y} = \sigma(w^\top x + b)$). Code listings are numbered per chapter (Listing 4.3 is the third listing in Chapter 4), and each chapter has a matching notebook in the companion repository, github.com/arkane-dev/book-companion-notebooks.

Exercises are tagged by kind, not just difficulty:

- **[W] Warm-up**: checks understanding of a definition or fact; a few minutes.
- **[A] Applied**: apply the chapter's method to new data or parameters; requires running code.
- **[E] Extension**: combines this chapter with prior material, or probes a trade-off boundary; open-ended.
- **[R] Research**: requires reading outside the chapter; discussion only in Appendix B, no single answer.

Prerequisites Map

This volume assumes comfort with Python (functions, loops, classes, `pip`) and the command line, but no prior statistics, calculus, or machine learning background. Chapters 1-5 build those foundations. Chapters are meant to be read in order on a first pass; later chapters assume earlier ones.

1 History & Landscape

For most of the 20th century, making predictions from data required a domain expert with intuition built over decades, or a statistician with a specific model already in mind. Neither scaled. The question the last 80 years of AI and data science have been trying to answer is: can machines learn patterns from data on their own, without being told the rules? Before we start to answer that question, we need to orient ourselves. This chapter is our map. It spans time and space: where the field came from, what it looks like today, and how the roles in it fit together. It's the grounding the rest of this book relies on.

This chapter has no AWS SageMaker Notes sidebar. We're making a map, not doing the work. The cloud-specific material begins at Chapter 2, once there's something concrete to deploy.

1.1 Motivation

Data science isn't a single discipline. It's the intersection of three older fields: **statistics** (making inferences from data under uncertainty), **computer science** (building systems that process information efficiently), and **domain expertise** (knowing enough about a problem to ask the right questions). None of these are new. Statistics has been around since at least 1662, when John Graunt and Sir William Petty analysed London's mortality rolls. Computer science arrived in 1936, when Alan Turing introduced the concept of the Turing machine. Domain expertise is as old as humanity. What is new is having all three at scale, at the same time, with cheap compute. Almost every technique in this book has a simpler ancestor. The "new" ideas in deep learning and large language models are extensions of ideas that are decades old.

1.2 Exposition

1.2.1 The Three Eras

Era 1 — Rules and Expert Systems (1950s-1980s). The first AI systems tried to encode human knowledge as rules: *if symptom A and symptom B, then diagnosis C.* The research focus of the era matched that structure. Engineers spent their time writing the

rules themselves, one `if/then` at a time. They didn't train a system to find the rules. These worked in narrow domains: chemistry (DENDRAL), medical diagnosis (MYCIN), computer configuration (XCON). But they failed to scale. Writing rules for everything a domain expert knows takes longer than training that expert. The insight that ended this era: the bottleneck isn't compute or algorithms. It's **knowledge acquisition**. Complex real-world domains can't be hand-coded. They have to be learned from data. Two names got fixed in place during this era. John McCarthy coined the term "artificial intelligence" for the 1956 Dartmouth Conference. That workshop is usually credited with founding AI as a research field. "Data science" arrived later in the same era. Danish computer scientist Peter Naur used it in his 1974 book *Concise Survey of Computer Methods*, as a synonym for "datalogy," a name he had proposed in the 1960s as a replacement for "computer science." This doesn't mean deep technical work wasn't happening. The world wasn't ready yet. The first neural networks were built during this era, on groundwork laid even earlier. Warren McCulloch and Walter Pitts proposed the first mathematical model of a neuron in 1943. Pure theory, no hardware. Frank Rosenblatt turned that idea into something real: the perceptron. He first simulated it in software in 1957, then built it as dedicated hardware, the Mark I Perceptron, demonstrated publicly in 1960. It read images through a 20x20 grid of photocells, making it one of the earliest computer vision systems. It was a single-layer network, not the multi-layer perceptron architecture used today. Stacking layers and training them with backpropagation didn't arrive until the 1980s. But it's still part of the architecture inside Large Language Models.

Era 2 — Statistical Machine Learning (1980s-2010s). Instead of encoding rules, the field pivoted to learning patterns from labelled examples. That pivot produced the algorithms that still dominate production ML today. Paul Werbos first described backpropagation in 1974. Rumelhart, Hinton, and Williams rediscovered it in 1986, and that made neural networks trainable. Support Vector Machines and AdaBoost, the first practical boosting method, both arrived in 1995. LSTMs came in 1997: recurrent networks that could learn dependencies across long sequences, which earlier recurrent networks could not. Random Forests and gradient boosting both arrived in 2001. Gradient boosting is the direct ancestor of XGBoost and LightGBM. The research focus moved with these tools, from writing logic to crafting features. That meant deciding which measurements, a pixel intensity, a transaction amount, a word count, predict the outcome, then feeding those into one of the algorithms above. Most practitioners at the time called this work "data mining": finding patterns already sitting in large databases rather than encoding them by hand. The field's own name caught up with the work late in this era. Statistician C. F. Jeff Wu argued for renaming statistics to data science in a 1997 lecture. William S. Cleveland made the case formally in a 2001 paper, proposing it as an expanded, independent field built on statistics' technical core. This era had self-driving cars earlier than most people realise. Carnegie Mellon's NavLab 1 drove itself down the road in 1986, steering by camera and onboard computer. A few years later, the same lab's ALVINN system went further. It threw out hand-written steering rules. In their place it used a neural network, trained by backpropagation, that learned to steer

straight from camera images. The limitation of this era was data and compute. Neural networks existed. Training them on anything interesting took too long.

Era 3 — Deep Learning and Foundation Models (2012-now). In 2012, a deep convolutional neural network called AlexNet won the ImageNet competition. It won by a margin that shocked the computer vision community. The ingredients had been assembling quietly for years: large labelled datasets, GPU compute, better training techniques. What followed was fast. GANs arrived in 2014. The Transformer architecture arrived in 2017 ("Attention Is All You Need"), replacing recurrence with attention and enabling massive training parallelism. BERT followed in 2018. GPT-3 landed in 2020 at 175 billion parameters, with emergent few-shot learning. ChatGPT arrived in 2022, combining instruction fine-tuning with RLHF. From 2023 onward came multimodal models, long context windows, agentic systems, and open-source models competitive with proprietary ones. The research focus shifted again. Instead of hand-crafting features, the model learns its own internal representations end-to-end, straight from raw pixels or tokens. What's left for a human to do is prepare and curate the data it learns from at scale. Feature engineering moved up, from the individual column to the whole dataset. The key insight of this era: **If in doubt, do as Texas does, and make it bigger**. Models trained on enough data with enough compute develop capabilities nobody explicitly trained them for. Nobody fully understands why.

1.2.2 The Landscape Today

Data science today is not a bunch of data scientists sitting around doing research. It's a cluster of specialisations with a shared toolbox:

Role	Core focus	Where they live
Data Analyst	Understanding what the data says now	BI tools, SQL, visualisation
Data Engineer	Building pipelines that move and store data	Spark, Kafka, dbt, cloud warehouses
Data Scientist	Building models that predict or explain	sklearn, XGBoost, PyTorch
ML Engineer	Making models work reliably in production	MLOps, serving, monitoring
Research Scientist	Advancing the state of the art	Papers, novel architectures
AI/ML Solutions Architect	Designing systems that use all of the above	You, after this book

As a Solutions Architect, your goal isn't to be the best at any one of these. You're a generalist. Your goal is to understand each area well enough to design systems that connect them. You also need to hold a credible conversation with each type of specialist. The AI stack most enterprise projects sit inside runs, roughly bottom to top, through six layers: raw, cleaned, labelled, and embedded data; then data engineering (pipelines, warehouses); then classical ML (XGBoost, sklearn, forecasting); then foundation models (GPT-4, Claude, Llama); then fine-tuned or prompted foundation models; then applications (chatbots, copilots, agents). Foundation models are increasingly a commodity. The differentiation is in data quality, fine-tuning, and the operational layer around the model. That operational layer is most of this book.

The table above is the technical side of the landscape. Every project also has a business side: account teams, strategists, product managers, and executive sponsors. They're the people asking what problem is worth solving and whether it's worth the budget, regardless of which company or industry it happens in. As a Solutions Architect, that's part of your job description too. You translate between the technical roles above and this business side, in both directions. You own the feasibility conversation when the two disagree. A data scientist's responsibility usually stops at translating a business ask into a measurable modelling problem. The ongoing relationship with the business side, the trade-offs, the expectation-setting, is not typically theirs to own. It is yours.

1.3 Trade-offs and Limits

Deep learning is better at unstructured data: images, text, audio, video. It's also better for tasks where the relevant features aren't known in advance, and for tasks with massive amounts of training data. Classical ML still wins on tabular data with clear features, most enterprise datasets. It wins in low-data regimes too, hundreds or thousands of examples rather than millions. It wins in situations requiring interpretability: regulated industries, medical, financial. And it wins on speed and cost. A gradient-boosted tree runs on a laptop. Kimi K3 does not.

Each approach has its own ceiling, not just a preference. Deep learning's limit is compute and data. Without enough of both, a large network won't beat a simpler model. It often won't even train stably. Classical ML's limit is the human doing the feature engineering. It can only be as good as the features fed into it. And hand-crafting those features doesn't scale to raw images, audio, or free text the way deep learning does. Foundation models' limit is their own generality. Trained to do everything reasonably well, they need fine-tuning, prompting, or retrieval on top before they do one specific enterprise task well. That adaptation step carries its own cost.

One of the most common mistakes in enterprise AI today: reaching for the biggest, most complex solution when a simpler one works. A foundation model, when a well-tuned XGBoost

model would be cheaper, faster, more accurate, and more explainable. A multi-agent, LLM-orchestrated mess, when a static, deterministic pipeline is all that's needed. Developing the judgement for which situation you're in is a running thread through this entire book.

1.4 Procedures

Before any of the ideas above become useful, we need to add tools to our toolbox. The procedures below are the first-steps recipes this book returns to constantly. Every later chapter's Worked Example assumes them.

Procedure 1.1 (Load a Dataset and Do a First Inspection). *When to use:* every time a new dataset is encountered. Never skip this.

Listing 1.1 The four inspection commands to run on any new dataset, every time

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.shape)        # (rows, columns)
df.head()               # first 5 rows
df.info()                # column names, types, non-null counts
df.describe()            # summary stats for numeric columns
```

What to look for: column types that don't match expectations, like a date column loaded as `object`. Missing values (`df.isnull().sum()`). Numeric ranges that don't look sensible. Wildly large or small values often mean an encoding error, not a real observation. Investigate a person with a negative age, or someone born in 1792.

Procedure 1.2 (Select and Filter Rows and Columns). *When to use:* narrowing data down to the subset being analysed.

Listing 1.2 Selecting columns and filtering rows by one or more conditions

```
# one column, as a Series
df['column_name']

# multiple columns
df[['col_a', 'col_b']]

# filter by condition
df[df['year'] > 2010]
```

```
# AND - use & and |
df[(df['year'] > 2010) & (df['subfield'] == 'deep_learning')]
```

Procedure 1.3 (Compute Grouped Summary Statistics). *When to use:* comparing a metric across categories.

Listing 1.3 Grouped aggregation, the workhorse of exploratory analysis

```
df.groupby('category')['value'].mean()

df.groupby('year').agg(
    total=('papers', 'sum'),
    average=('papers', 'mean'),
    count=('papers', 'count'),
)
```

Procedure 1.4 (Create a Derived Column). *When to use:* feature engineering, or calculating a derived metric for a chart.

Listing 1.4 Common derived-column patterns: arithmetic, year-on-year change, mapping

```
df['ratio'] = df['col_a'] / df['col_b']
df['yoy_change'] = df['value'].pct_change() * 100
df['label'] = df['value'].apply(
    lambda x: 'high' if x > 100 else 'low'
)
```

Common mistake: `pct_change()` returns `NaN` for the first row (there's no prior value to compare against). That's expected, not a bug. Drop it with `.dropna()` or fill with `.fillna(0)`, depending on what the chart needs.

Procedure 1.5 (Plot and Annotate a Line Chart). *When to use:* showing how a value changes over an ordered variable, usually time, and marking a specific event on it.

Listing 1.5 A minimal annotated line chart, saved before it is shown

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
plt.plot(
    df['year'], df['value'],
    label='My Series', color='#4E79A7', linewidth=2,
)
plt.axvline(x=2017, color='grey', linestyle='--', linewidth=1.5)
```

```
plt.text(
    2017.2, plt.ylim()[1] * 0.9,
    'Key event', fontsize=9, color='grey',
)
plt.xlabel('Year')
plt.ylabel('Value')
plt.legend()
plt.tight_layout()
# savefig BEFORE show()
plt.savefig(
    'output.png', dpi=150, bbox_inches='tight',
)
plt.show()
```

1.5 Worked Example

The tutorial notebook behind this chapter builds two things: a visual timeline of the milestones above, and a real trend analysis of machine learning research growth. Both use nothing but Procedures 1.1-1.5.

The timeline starts as data the reader defines by hand. That's practice building structured records in Python before loading anything external:

Listing 1.6 Milestones as structured data before they become a chart

```
RULES = "Rules & Expert Systems"
STATS_ML = "Statistical ML"
DEEP_LEARNING = "Deep Learning & Foundation Models"

milestones = [
    {"year": 1950, "label": "Turing Test proposed", "era": RULES},
    {"year": 1956, "label": "Dartmouth Conference", "era": RULES},
    {"year": 1986, "label": "Backpropagation rediscovered", "era": STATS_ML},
    {"year": 2001, "label": "Random Forests", "era": STATS_ML},
    {"year": 2012, "label": "AlexNet wins ImageNet", "era": DEEP_LEARNING},
    {
        "year": 2017,
        "label": "Transformer architecture ('Attention Is All You Need')",
        "era": DEEP_LEARNING,
    },
    {"year": 2022, "label": "ChatGPT", "era": DEEP_LEARNING},
]
```

```
df_milestones = pd.DataFrame(milestones)
n_eras = df_milestones['era'].nunique()
print(f"{len(df_milestones)} milestones across {n_eras} eras")
```

Procedure 1.5 plots this as points on a timeline, with era-coloured backgrounds and per-point annotation. It becomes a single figure spanning the same three eras this chapter opened with. It's the same structured-data-then-visualise pattern used for feature engineering and distribution plots throughout the book.

The second half loads real arXiv paper counts by subfield and year, 1993 to 2024, pulled from the arXiv API. arXiv has no "deep learning" category, so each count is the number of papers that year whose abstract mentions one of the subfield's key phrases, such as "deep learning" or "random forest". A paper can count in more than one subfield. Then it asks a question instead of plotting it: **what year did deep learning paper volume first exceed classical ML?**

Listing 1.7 Answering a specific question from data, not just plotting it

```
df['dl_vs_classical'] = df['deep_learning'] - df['classical_ml']
# .iloc[0] is the earliest year only if rows are in year order
df = df.sort_values('year')
crossover = df[df['dl_vs_classical'] > 0].iloc[0]

year = int(crossover['year'])
classical = crossover['classical_ml']
deep = crossover['deep_learning']

print(f"Deep learning overtook classical ML in: {year}")
print(f"  Classical ML papers that year: {classical:,}")
print(f"  Deep learning papers that year: {deep:,}")
```

The crossover falls in **2015** (classical ML at 794 papers, deep learning at 933). Three years after AlexNet. Deep learning then grew fast, to more than 5 papers for every classical ML paper by 2020. The ratio has held near 5 since: 4.8 to 1 in 2024. Classical ML kept growing steadily the whole time. The newest surge is language models. Papers mentioning them more than doubled in both 2023 and 2024, and overtook deep learning in 2024.

One gotcha worth keeping: computing year-on-year growth with `pct_change()` (Procedure 1.4) can blow up. If a series starts at exactly zero, the first nonzero value produces an infinite percentage. That's a real `inf` in the output. Not a bug. It's the kind of artefact `nlargest()` will happily rank first if it isn't excluded by hand. Once that artefact is excluded, the NLP series' largest growth year is 2000 (350%), a low-base effect: 6 papers to 27. The first big jump on a large base is 2023 (169%, from 4,253 papers to 11,447), the year after ChatGPT's release. 2024 follows at 112%.

1.6 History and Further Reading

Burkov, Ch. 1, gives a crisp introduction to what machine learning is and isn't (Burkov 2019). Murphy's Introduction, Ch. 1, gives the formal framing of supervised, unsupervised, and reinforcement learning (Murphy 2022). "Attention Is All You Need" started the transformer era. It's worth skimming even on a first read of this book (Vaswani et al. 2017). Domingos' "A Few Useful Things to Know About Machine Learning" is a classic essay. It's still one of the best single pieces on ML intuition (Domingos 2012). For a broader frame around everything in this book, look at Sayama's *Introduction to the Modeling and Analysis of Complex Systems*. It situates machine learning within the wider study of complex systems, rather than treating it as a standalone discipline. That's a framing worth having before the field-specific chapters that follow narrow the focus (Sayama 2024).

1.7 How This Connects

This chapter builds on the systems thinking a Solutions Architect already has. It enables everything that follows. We've created the map the rest of the book explores. One common mistake: treating AI, machine learning, and deep learning as synonyms. They're nested subsets. AI is the broadest term. Machine learning is a subset of it. Deep learning is a subset of machine learning that specifically uses neural networks.

1.8 Exercises

1. **[W]** Name the era (Rules & Expert Systems / Statistical ML / Deep Learning & Foundation Models) each of the following belongs to, and give one reason it belongs there: (a) Random Forests; (b) BERT; (c) MYCIN.
2. **[W]** Explain in one or two sentences why "AI," "machine learning," and "deep learning" are not synonyms, using the nesting relationship from this chapter.
3. **[A]** Using Listing 1.6 as a starting point, add three milestones of your choosing from outside this chapter's list (they can be from any era). Then, using Procedure 1.2, determine how many of the resulting milestones fall after the year 2017.
4. **[A]** Reproduce Listing 1.7's logic to find the year reinforcement learning paper volume first exceeded classical ML paper volume. Report the ratio between them in 2024.
5. **[E]** A customer's data science team has 3,000 labelled examples of a tabular fraud-detection problem and is proposing to fine-tune a foundation model for it. Using this chapter's Trade-offs section, write the two or three sentences you'd say to them. Include what you'd suggest they try first.

6. [**E**] The chapter notes that `pct_change()` on a zero-to-nonzero transition produces an infinite percentage. Propose two different ways to handle this before ranking growth years with `nlargest()`. State a situation where each would be the wrong choice.
7. [**R**] Read the Domingos essay cited above. Pick one "useful thing" from it that this chapter did not mention. Explain how it would have changed the way Listing 1.7's finding was interpreted if applied to it.

1.9 Mental Model

Data science is applied statistics with better compute and worse theory. Machine learning is a way of fitting functions to data. Deep learning is what happens when those functions get very large and are trained on huge amounts of data. A foundation model is deep learning at a further extreme: even bigger networks, trained on even more data. It's general enough to be pointed at many different tasks instead of one. That's also why it sometimes does things nobody explicitly trained it to do. The rest is engineering. Keep this deflationary framing. It prevents being intimidated by complexity that is just implementation detail dressed up in jargon.

2 Business Problem Framing

When ML projects fail, it's not always because the model was bad. They often fail because the model was solving the wrong problem. A customer asks for "AI to improve our operations." Six months later, a team ships a churn model that scores customers accurately. But operations never plugged it into any workflow. Nothing changed. This chapter is about the translation step that prevents that failure mode: turning a vague business ask into a precisely defined ML problem statement, before anyone writes a line of code.

2.1 Motivation

The gap between "we want AI" and "we have a deployable ML system solving a defined business problem" is where Solutions Architects excel. Every ML project starts with a business problem. Your job is to translate that problem into a precisely defined ML task before touching data. That translation has three steps: understand the business goal, define the ML problem, and define success in measurable terms. These sound obvious. In practice, most projects skip or rush all three. The rest of this chapter shows what happens when they do.

2.2 Exposition

2.2.1 Understanding the Business Goal

The customer's stated ask is almost never the right ML problem. A few examples:

Customer says	What they need
"We want to predict churn"	Reduce revenue loss → need to identify *preventable* churn with enough lead time to act
"We want to understand our customers better"	Segment customers → unsupervised clustering, not a predictive model

Customer says	What they need
"We want a recommendation engine"	Increase conversion → rank items by probability of purchase per user
"We want to detect fraud"	Minimise fraud loss → anomaly detection with a specific false-positive budget

This isn't a data science problem. It's a translation problem. Requirements engineering solved a version of it decades before ML existed. The customer's stated ask is usually a solution, not a need: "we want a recommendation engine" describes a piece of software, not a business outcome. The real requirement sits one level up: increase conversion. The recommendation engine is one way to get there. Maybe not the best one. Digging out the real requirement, not the stated solution, is job zero.

A reliable way to do that: stop talking to just one person. Most ML engagements interview whoever asked for the project and treat that as "the customer's requirements." But almost every project has three distinct people or groups behind the ask. They rarely agree on what matters:

- **The sponsor** pays for the project and owns the business outcome. For a churn model, that's usually a VP of Customer Success with a revenue target.
- **The customer** accepts the deliverable and judges whether it solved the problem. That might be a Director of Retention who has to sign off before the model goes live.
- **The users** are the people who touch the model's output every day. For a churn model, that's the retention agents working the accounts the model flags.

Talk to only the sponsor, and the goal statement reads like a budget line: reduce churn, save money. Talk to only the users, and you'll hear the operational detail no one above them knows. Retention agents often ignore a raw risk score unless it comes with a reason and a suggested next action. A lone number gives them nothing to act on. Miss that, and the model can hit every accuracy target in the spec and still get ignored on the floor. Skipping the users is one of the most common ways an accurate model ships and changes nothing.

Once the goal is understood from all three angles, write it down using three prongs: **purpose**, **advantage**, and **measurement**.

- **Purpose**: what the project is trying to do, in plain language. Not "build a churn model," but "identify customers at risk of cancelling with enough lead time for a retention agent to intervene."
- **Advantage**: the business benefit, tied to money or a metric leadership already tracks. "Reduce the revenue lost to preventable cancellations."
- **Measurement**: how you'll know it worked, stated as a number. "1,500 prevented cancellations per month" is a measurement. "Improved retention" is not.

Purpose and advantage are usually easy to get agreement on. Measurement is where vague projects get caught: if nobody can state a number, the goal isn't defined yet, no matter how much everyone nodded along in the kickoff meeting.

This process matters because it determines what gets built. "Predict churn" leads to a classifier. "Reduce preventable revenue loss" leads to a classifier *plus* a scoring system *plus* an intervention workflow. The model alone is worthless without the others.

2.2.2 Defining the ML Problem

Once the goal statement exists, with purpose, advantage, and measurement all written down, five questions turn it into something a model can be built against.

What is the label? This is the thing you want to predict. It must be specific, measurable, and tied to a real event. "Customer satisfaction" is not a label. "Customer submits a cancellation request within 30 days" is.

What is the unit of analysis? What does one row in your training data represent? One customer, one transaction, one day per customer? Getting this wrong produces models that leak future information into the past, or that predict at the wrong granularity.

What features are available at prediction time? A model is only useful if you can run it in production. If your most predictive feature requires data that doesn't exist until after the event you're predicting, the model is useless. This is one of the most common mistakes in ML projects.

What is the prediction horizon? "Predict churn 30 days before it happens" is a different problem from "predict churn 3 days before." The former gives time to intervene. The latter not so much.

Is ML the right tool? ML is often unnecessary. Does a simpler rule already work well enough? Is there a pattern to learn, or is the process effectively random? Is there enough data? Appendix A's ML Suitability Checklist gives a fuller diagnostic.

2.2.3 Defining Success

This is the measurement prong from the goal statement, formalised: a business metric that's testable, not just a metric that sounds good. "Accuracy" is almost never the right metric in business contexts. If 95% of transactions are legitimate and 5% are fraud, a model that predicts "legitimate" for every transaction scores 95% accuracy and catches zero fraud. Useless. Business success metrics instead connect model outputs to business outcomes:

- **Fraud detection**: at 1% fraud prevalence, catch 80% of fraud (recall ≥ 0.8) while reviewing no more than 2% of transactions manually (precision ≥ 0.4 at threshold).

- **Churn prevention**: score the top 10% of at-risk customers. Intervention converts 15% of those, for a target of $\geq$ 1,500 prevented cancellations per month.
- **Recommendation**: increase click-through rate from 1.2% to 1.8% in an A/B test ($p < 0.05$).

Notice none of these are model metrics. They're business outcomes derived from model performance. A problem statement needs both.

2.2.4 The Precision-Recall Trade-off as a Business Decision

Every classification model produces a probability for each prediction, and a **threshold** above which you act. Moving the threshold trades precision against recall:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN}$$

A high, strict threshold flags something only when the model is confident. High precision, low recall, misses real cases. A low, permissive threshold flags anything that looks suspicious. High recall, low precision, many false positives. Which one is right isn't a data science decision. It's a business decision: what does a false positive cost, versus a false negative? The business owns that answer, not the model.

There's a formula version of this trade-off too: the **F-beta score**, a weighted variant of the familiar F1 score.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \times \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

β sets how much more recall matters than precision. Set $\beta = 1$ and it's the ordinary F1 score. That treats a false positive and a false negative as equally costly, which is rarely true in a real business. F-beta weights each missed positive β^2 times as heavily as each false alarm. So a practical starting point is $\beta = \sqrt{C_{FN}/C_{FP}}$. A missed fraud case at \$200 against a \$5 manual review, the same cost pair Procedure 2.4 uses, gives $\beta = \sqrt{40} \approx 6.3$: a score weighted heavily towards recall. This isn't a substitute for the threshold-cost sweep in Procedure 2.4. That sweep finds the actual optimal threshold, dollar for dollar. The F-beta score is a faster single number for comparing two candidate models before either one has a chosen threshold at all.

2.2.5 The ML Problem Type Taxonomy

Type	Question being answered	Label type	Example
Binary classification	Will this happen or not?	0/1, True/False	Churn, fraud, click
Multi-class classification	Which category?	One of N classes	Product category, sentiment
Regression	How much / when?	Continuous number	Revenue, time-to-event, price
Ranking	In what order?	Relative order	Search results, recommendations
Clustering	What natural groups exist?	None (unsupervised)	Customer segments, topic groups
Anomaly detection	Is this unusual?	None or sparse labels	Fraud, sensor failure, intrusion
Time series forecasting	What will happen next?	Future value	Demand, sales, traffic

Picking the wrong type means building the wrong thing. "Customer segments" sounds like clustering. But if churn labels already exist, supervised classification can find segments *that differ in churn rate*. That's far more actionable than an unsupervised grouping.

2.3 Trade-offs and Limits

Wrong framing kills ML projects in three recurring ways. **The label problem**: a team builds a model to predict "customer engagement score" because that's the metric on the dashboard. But the actual business goal is subscription renewals. Optimising engagement doesn't move renewals. Six months of work produces no ROI. **The leakage problem**: a churn model shows 99% accuracy in testing, then drops to 60% in production. That's because a feature like "days since last support ticket closed" is only populated *after* a customer has already churned. The model learned to detect the symptom, not the cause. **The horizon problem**: a churn model correctly predicts churn 3 days in advance. But the sales team's intervention process takes 5 days minimum. The model is technically correct and practically useless.

Each of these is a framing failure, not a modelling failure. That's why it happens before Chapter 11's classifiers ever enter the picture.

2.4 Procedures

Procedure 2.1 (Write a Structured ML Problem Statement). *When to use:* at the start of any ML engagement, before any data exploration or modelling. Fill in every field of the template in Appendix A's Problem Statement Template before proceeding. Leaving a field blank is a signal that the framing isn't finished.

Procedure 2.2 (Identify the Label). *When to use:* when the customer describes a goal but hasn't specified what to predict. Ask, in sequence. What event are we trying to predict? Turn "customer dissatisfaction" into "submits cancellation request." When does that event occur? This forces a time window, e.g. "within 30 days of billing cycle end." How do we know it happened? This forces measurability. What's the unit: per customer, per transaction, per day? Then verify the label exists in the data:

Listing 2.1 Verifying a candidate label exists and checking its class balance

```
import pandas as pd

df = pd.read_csv('churn_dataset.csv')
print(df['churned_within_30d'].value_counts())
print(df['churned_within_30d'].isnull().sum(), "missing values")
print(f"Positive rate: {df['churned_within_30d'].mean():.1%}")
```

Common mistake: accepting a label the customer has already computed (e.g. an "engagement score" column) without checking whether that computed quantity was the actual business goal.

Procedure 2.3 (Spot When ML Is Overkill). *When to use:* before committing to build a model. Check whether a simple rule already matches the required performance. Check whether the process is fundamentally random. Check whether there are roughly 1,000 labelled examples per class. A quick baseline makes this concrete. If a hand-written rule already clears the bar, ship the rule instead:

Listing 2.2 Baselining a hand-written rule before building a model

```
from sklearn.metrics import classification_report

rule_predictions = (
    (df['num_support_tickets_90d'] > 2)
    & (df['payment_failures_6m'] > 0)
).astype(int)
```

```
print(classification_report(
    df['churned_within_30d'], rule_predictions
))
```

Procedure 2.4 (Translate Precision vs. Recall into a Business Decision). *When to use:* whenever a customer asks "how accurate is the model?" Redirect to this framing instead. Attach a dollar cost to a false positive and a false negative. Then sweep thresholds to find the cost-minimising point:

Listing 2.3 Sweeping decision thresholds to minimise business cost, not maximise accuracy

```
import numpy as np
from sklearn.metrics import confusion_matrix

def business_cost(y_true, y_pred, fp_cost, fn_cost):
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    return fp * fp_cost + fn * fn_cost

# e.g. $5 to manually review, $200 for a missed fraud case
fp_cost, fn_cost = 5, 200
thresholds = np.linspace(0.1, 0.9, 50)
costs = [
    business_cost(
        y_test, (probs > t).astype(int), fp_cost, fn_cost
    )
    for t in thresholds
]
optimal_threshold = thresholds[np.argmin(costs)]
```

Rule of thumb: if a false positive is the expensive mistake (e.g. wrongful fraud flag), raise the threshold. If a false negative is the expensive mistake (e.g. missed cancer diagnosis), lower it.

Procedure 2.5 (Check for Data Leakage). *When to use:* before finalising a feature list. Leakage produces models that look excellent in testing and fail in production. List every candidate feature. Ask whether it could encode information that wouldn't exist at prediction time. Then sort features by correlation with the label. Anything above roughly 0.9 warrants investigation before use.

2.5 Worked Example

Three Solutions Architect scenarios illustrate how differently "we want AI" cashes out once it's framed. A telco says: *"We're losing customers and we want to use AI to fix it."* Working through Procedure 2.2 turns this into: predict `churned_within_30_days` per customer at end of billing cycle, 30 days ahead of the churn date. The model uses account and support-ticket history available at that point. That's a binary classification problem. A retailer's *"we want to personalise our homepage"* becomes a ranking problem over (user, item) pairs with a 200ms latency budget. A manufacturer's *"predict when machines will break down"* becomes multi-class classification over failure modes, on hourly-aggregated sensor windows, 48 hours ahead of failure. Three "AI" requests, three structurally different ML problems. None of them obvious from the customer's original words.

Continuing with the telco scenario, Listing 2.1's schema is used to generate a synthetic 10,000-row churn dataset. It's inspected for label distribution before any modelling begins:

Listing 2.4 Building and inspecting a correctly-schemaed churn dataset

```
feature_cols = [
    'months_as_customer', 'monthly_spend',
    'num_support_tickets_90d', 'days_since_last_login',
    'num_products', 'payment_failures_6m',
]
label_col = 'churned_within_30d'

X = df[feature_cols]
y = df[label_col]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=RANDOM_SEED
)
```

The dataset comes out imbalanced. Far more non-churners than churners in any given month. That's realistic. Chapter 13 (Cross-Validation & Model Selection) and Chapter 16 (Model Evaluation) return to exactly this imbalance with stratified splits and class weighting. For now, the point is simpler: label separated from features, no leakage, class balance preserved across the split. That is the scaffold every later model in this book is built on.

Procedure 2.4's threshold sweep needs one thing this chapter hasn't built yet: a model that outputs a probability. A quick, class-weighted logistic regression on the schema above supplies one, purely to make the mechanics concrete. This isn't real model-building. Chapter 11 covers that properly.

Listing 2.5 A quick baseline classifier, just to generate probabilities for the threshold-cost sweep

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(class_weight='balanced', random_state=RANDOM_SEED)
clf.fit(X_train, y_train)
probs = clf.predict_proba(X_test)[:, 1]
```

Run Listing 2.3's sweep against these probabilities, with `fp_cost=5` and `fn_cost=200`:

Threshold	False positives	False negatives	Cost
0.3	1,869	0	$9,345
0.5	651	75	$18,255
0.7	37	123	$24,785

0.3 wins by a wide margin, though it's an extreme case: this model's probabilities never drop below about 0.32, so a threshold of 0.3 flags essentially every customer in the test set. It still catches all 131 churners, and at $5 per false positive against $200 per missed churner, flagging everyone costs less than missing anyone. The full 50-point sweep finds the true minimum a little higher, at 0.345 ($9,185). Push the threshold to 0.7 instead, and the model misses 123 of those 131 churners to avoid a small number of false alarms: exactly the mistake a business that loses $200 per missed churner can't afford. This is Procedure 2.4's rule of thumb in action. The expensive mistake here is the missed churner, so the threshold belongs low.

AWS SageMaker Equivalent

```
import boto3

s3 = boto3.client('s3')
s3.upload_file(
    'churn_dataset.csv',
    'your-bucket-name',
    'datasets/churn/churn_dataset.csv',
)

# Register in SageMaker Feature Store
# (Book 2, Chapter 26 covers this in detail)
# import sagemaker
```

```
# feature_store = sagemaker.feature_store.FeatureStore(
#     sagemaker_session=session
# )
```

When to prefer SageMaker Feature Store: when multiple teams or models need to share the same feature definitions and require point-in-time correct feature retrieval. That avoids leakage at scale, not just within a single project.

2.6 History and Further Reading

Burkov, Ch. 1–2, gives the formal definition of supervised learning and the anatomy of a dataset (Burkov 2019). Thakur, Ch. 1, takes a practical approach to setting up an ML project from scratch (Thakur 2020). Hadley Wickham's "The Data Science Process" argues for an iterative framing. It's worth reading against this chapter's three-step sequence. See Exercise 2.7 (Wickham, n.d.).

> **i** If Requirements Gathering Interests You
>
> Everything above in "Understanding the Business Goal" (the sponsor/customer/user split, writing a goal as purpose, advantage, and measurement) is adapted from requirements engineering. That's a discipline with decades of hard-won practice on exactly this translation problem. Suzanne Robertson and James Robertson's *Mastering the Requirements Process* is the standard reference. It's worth buying outright if this translation step interests you more than the modelling that follows it (Robertson and Robertson 2012).

2.7 How This Connects

This chapter builds on Chapter 1, which surveys what kinds of models exist. It enables essentially everything after it. Most directly, Chapter 10 (The Data Science Lifecycle) builds the same framing into a repeatable process. It is often confused with requirements gathering. But framing is not just capturing what the customer wants. It is actively translating that want into a technically precise ML problem statement.

2.8 Exercises

1. **[W]** A customer says "we want to predict customer satisfaction." Explain in one sentence why this is not yet a usable label, and propose a candidate label that would be.

2. [**W**] For each of the following, name the correct entry from the Section 2.2.5 taxonomy: (a) grouping customers into segments with no target variable; (b) predicting the dollar value of a customer's next order; (c) predicting whether a machine will fail in the next 48 hours.
3. [**A**] Using the churn dataset schema from Listing 2.4, propose a new candidate feature `days_since_last_price_increase`. Using Procedure 2.5, determine whether it is safe to use. Justify your answer.
4. [**A**] Using Listing 2.5's probabilities and Listing 2.3's sweep, verify the total business cost at thresholds 0.3, 0.5, and 0.7 with `fp_cost=5`, `fn_cost=200`. State which threshold minimises cost, and why.
5. [**E**] The chapter argues accuracy is rarely the right metric. Construct a scenario, distinct from fraud or churn, where accuracy is an acceptable metric. Explain why the class-imbalance argument doesn't apply there.
6. [**E**] A customer asks for "AI to detect anomalies in manufacturing sensor data" but has zero historical examples of past failures. Using the ML Suitability Checklist in Appendix A, decide whether ML is appropriate here. If not, propose what to build instead.
7. [**R**] Read the Wickham piece cited above. Compare its iterative framing to this chapter's three-step sequence (understand the goal → define the ML problem → define success). Where do they agree, and where does iteration break the sequential picture this chapter presents?

2.9 Mental Model

The ML problem statement is a contract. It specifies the prediction target, the unit of analysis, the features available at prediction time, the horizon, and the success metrics. If any one of these is ambiguous, the project fails. Not because of bad algorithms. Because of misaligned expectations. Write the contract before building anything.

3 Maths Foundations: Statistics, Probability & Linear Algebra

Every machine learning algorithm is, underneath its API, a small set of mathematical ideas wearing a Python trenchcoat. Those ideas are a probability that expresses uncertainty, a statistic that summarises a sample, or a matrix operation that transforms a dataset. None of this requires re-deriving proofs from first principles. But it does require some fluency. You need to read an equation without flinching and notice when a model's assumptions don't match the data. You also need to explain to a customer why a result means what it means rather than what they hoped it meant. This chapter covers the three branches of mathematics that recur throughout the rest of this book: probability, statistics, and linear algebra. Not as abstract theory. As the specific vocabulary the following seventeen chapters assume. By the end, a single worked thread ties them together: the closed-form solution to linear regression. That's simultaneously a statistics problem (minimise squared error), a linear algebra computation (solve a matrix equation), and the first model most of this book's later classifiers and regressors generalise.

3.1 Motivation

Three older fields fuse into what we call data science. **Probability** is the calculus of uncertainty. **Statistics** is the practice of summarising data and drawing defensible inferences from it. **Linear algebra** is the language for representing and transforming structured data at scale. Most classifiers compute a probability underneath, even when the output is dressed as a class label. Most of exploratory data analysis is applied statistics, whether or not the word "statistics" ever appears in the code. And every dataset is, mechanically, a matrix. Every model operation, from a single linear regression to a hundred-layer neural network, is some sequence of matrix operations applied to that matrix. Skipping this chapter doesn't make these ideas go away. It just means meeting them later, embedded in a library, without the vocabulary to reason about what they, and by extension you, are doing.

3.2 Exposition

3.2.1 Probability

A probability is a number between 0 and 1 that quantifies how likely an event is. $P(A) = 1$ means A always happens. $P(A) = 0$ means it never does. $P(A) = 0.3$ means it happens 30% of the time, in the long run, across repeated trials. **Conditional probability**, written $P(A \mid B)$, is the probability of A given that B is already known to be true. This is the most important operation in machine learning. "Given these features, what is the probability of this outcome?" is a conditional probability question, whether the model computing it is a logistic regression or a hundred-billion-parameter transformer.

Bayes' Theorem ties conditional probabilities together and is probably the most important equation in the field:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

In ML terms, B is the evidence you observe (features) and A is the hypothesis you care about (a label). $P(A)$ is the **prior**: what you believed before seeing any evidence, typically the base rate at which A occurs in the population. $P(B \mid A)$ is the **likelihood**: how probable the evidence is, assuming the hypothesis is true. $P(A \mid B)$, the **posterior**, is what you want: the updated probability of the hypothesis after accounting for the evidence.

A concrete, and deliberately uncomfortable, example makes the theorem's practical bite obvious. Suppose a disease has 1% prevalence in a population ($P(\text{disease}) = 0.01$). A diagnostic test has 99% sensitivity ($P(\text{positive} \mid \text{disease}) = 0.99$) and 95% specificity ($P(\text{negative} \mid \text{healthy}) = 0.95$, so $P(\text{positive} \mid \text{healthy}) = 0.05$). If someone tests positive, what's the probability they have the disease? Expanding the denominator with the law of total probability, $P(B) = \sum_i P(B \mid A_i)\, P(A_i)$, gives

$$P(\text{disease} \mid \text{positive}) = \frac{0.99 \times 0.01}{(0.99 \times 0.01) + (0.05 \times 0.99)} \approx 0.167$$

Only 16.7%, despite a test that looks 99% accurate on paper. The base rate, the disease's rarity, dominates the result. The population of healthy people is so much larger than the population of diseased people that even a small false-positive rate among the healthy majority swamps the true positives from the rare minority. This is the mechanism behind the "high accuracy but useless" trap discussed for class-imbalanced classifiers in Chapter 2: every time a model is evaluated on data where the positive class is rare, that evaluation is implicitly a Bayes'-theorem calculation.

3.2.2 Common Probability Distributions

A distribution describes the shape of randomness expected in a variable: which values are likely, which are rare, and how spread out they are. Four distributions recur constantly in ML work:

The **Gaussian (Normal)** distribution is the familiar bell curve, parameterised by mean μ and standard deviation σ, with density

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

It arises whenever many small, independent effects sum together. That's the Central Limit Theorem's signature. It's why measurement error, model residuals, and many naturally occurring quantities are roughly Gaussian. Linear regression's maximum-likelihood derivation assumes Gaussian residuals.

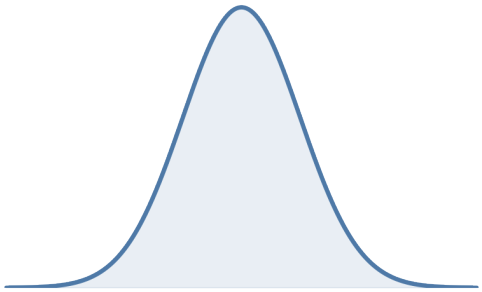


Figure 3.1: The Gaussian: symmetric, bell-shaped, mass concentrated around the mean.

The **Bernoulli** distribution describes a single binary outcome: 1 with probability p, 0 with probability $1 - p$. Every binary label in a classification problem (churned or not, fraud or not) is a Bernoulli random variable. Logistic regression is built explicitly around this assumption.



Figure 3.2: The Bernoulli: two outcomes, one bar per outcome, heights set by p.

The **Binomial** distribution is the sum of N independent Bernoulli trials ("how many successes in N attempts?"), with mean Np and variance $Np(1-p)$.

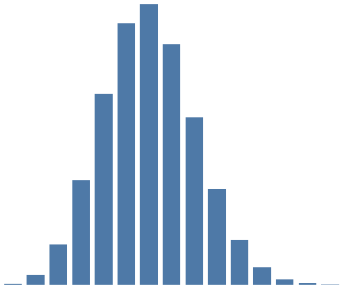


Figure 3.3: The Binomial: a discrete, bell-like shape over the count of successes.

Compare that shape to the Gaussian a few paragraphs back. The resemblance isn't accidental. A Binomial is a sum of independent Bernoulli trials. Summing many small independent effects is exactly the Central Limit Theorem's territory. Once Np and $N(1-p)$ are both comfortably above about 5, as they are in Figure 3.3, the Binomial looks Gaussian because it's converging towards one. This is old news: Abraham de Moivre proved it in 1733, for the special case $p = 0.5$, long before anyone had named it the Central Limit Theorem. Pierre-Simon Laplace generalised the result to any p in 1812. The result carries both their names today, the **De Moivre-Laplace theorem**: one of the oldest pieces of mathematics still doing daily work in this book.

The **Poisson** distribution describes counts of rare, independent events within a fixed interval (fraud events per day, machine failures per week), governed by a single rate parameter λ, with both mean and variance equal to λ.

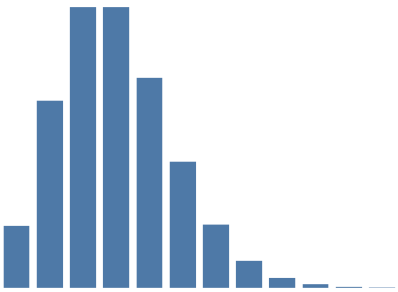


Figure 3.4: The Poisson: right-skewed, a sharp rise and a long thinning tail.

Choosing the wrong distributional assumption is a common modelling error. One example is fitting ordinary linear regression to count data with a long right tail, where a Poisson or negative-binomial model would fit far better. Another is assuming Gaussian residuals

when the true error distribution is heavy-tailed. The distribution is a claim about the data-generating process. The model's behaviour inherits that claim's correctness or wrongness.

3.2.3 Statistics

Descriptive statistics summarise what a dataset looks like without invoking any inferential machinery. Most of this you already know from school, even if the vocabulary is new. The **mean** is the arithmetic average (what most people call "the average") and is sensitive to outliers. The **median** is the middle value and is robust to them. When mean and median diverge by a lot, the underlying distribution is skewed.

> **i** Where's the Mode?
>
> You learned three averages in school: mean, median, and mode. Mean and median get most of the attention here. Mode, the most frequent value, rarely does, for good reason. On continuous data it's often ill-defined: every exact value occurs with roughly zero probability, so an empirical mode depends entirely on how the data gets binned. Mean and median don't have that problem. When it comes to optimisation: the mean is the single value that minimises the sum of *squared* deviations from it, and the median minimises the sum of *absolute* deviations. Mode isn't the answer to any such minimisation problem. It's just the most common value, and a distribution can have several of those, or none worth naming. Mode still earns its keep in specific spots: imputing missing categorical values, a "predict the most common class" baseline, and peak-finding in density estimation. It just isn't part of this cluster of summary statistics, because it isn't built the same way.

Variance, $\sigma^2 = \frac{1}{n}\sum_i (x_i - \mu)^2$, measures average squared deviation from the mean. That divides by n, the population version. pandas' `.std()` and `np.cov` divide by $n-1$ instead, the sample version, which matters on small samples. **Standard deviation**, $\sigma = \sqrt{\sigma^2}$, restores the original units. **Covariance** measures how two variables move together: positive when they rise and fall in tandem, negative when one rises as the other falls. But its magnitude is scale-dependent, so it's hard to interpret directly. **Correlation** (Pearson's r) normalises covariance into the fixed range $[-1, 1]$:

$$r = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

making it comparable across variables measured in entirely different units.

Hypothesis testing is the framework for deciding whether an observed pattern reflects something real or could plausibly be random chance. The procedure: assume no effect exists (the null hypothesis H_0). Compute how surprising the observed data would be under that assumption. Reject H_0 if the data is unlikely enough. The **p-value** is the probability, under

H_0, of observing data at least as extreme as what was observed. $p < 0.05$ is the conventional cutoff for "statistically significant." That's a threshold chosen by convention, not derived from any deep principle. And it's the most misused number in science.

Push on that threshold and it stops meaning much. Try enough subgroups, metrics, or stopping points on the same dataset, and something will clear $p < 0.05$ by chance alone. That practice has a name: **p-hacking**, often unconscious rather than deliberate. That practice is a central cause of what's now called the replication crisis. In 2015, the Open Science Collaboration tried to redo 100 published psychology studies and found only 36% produced a statistically significant result the second time. The original studies weren't necessarily fraudulent. Most were the predictable output of researchers making reasonable-looking choices, one p-value at a time, until something crossed the line.

The trap: a small p-value means the data would be surprising if there were no effect, assuming you haven't gone rooting around in the undergrowth for it. But it says nothing about whether that effect is *large* or *practically meaningful*. With enough data, a model improvement from 0.9000 to 0.9001 accuracy can register $p < 0.001$ while being irrelevant to any business decision. A p-value should never be reported alone. It needs an accompanying **effect size** (such as Cohen's d, the standardised difference between two group means) and, ideally, a confidence interval, so that statistical significance and practical significance can be evaluated separately.

3.2.4 Linear Algebra

> **i** Reading the Notation: $\mathbb{R}$ and $\in$
>
> Two symbols show up constantly from here on. $\mathbb{R}$ is the set of all real numbers: any number on the number line, positive or negative, with or without decimals. $\mathbb{R}^7$ means "ordered lists of 7 real numbers," so a customer vector living in $\mathbb{R}^7$ just means it has 7 numeric features. $\in$ means "is an element of," or more plainly, "belongs to" or "is a member of." $X \in \mathbb{R}^{N \times M}$ reads as "X belongs to the set of all real-valued matrices with N rows and M columns." That's a formal way of saying "X is an $N \times M$ matrix of real numbers." Neither symbol does anything exotic. They're set-theory notation borrowed to say precisely what shape and type of number is being talked about.

A **vector** is an ordered list of numbers. In ML, one row of a dataset (one data point) is a vector. A customer described by seven features (tenure, spend, ticket count, and so on) is a single point in $\mathbb{R}^7$. In code, a vector is a numpy array: fixed-length, one numeric type, built to support the arithmetic below directly.

> **A List Is Not a Vector**
>
> A Python list looks like a vector but doesn't behave like one. `[1, 2, 3] + [4, 5, 6]` concatenates into `[1, 2, 3, 4, 5, 6]`, not `[5, 7, 9]`. A numpy array does the elementwise thing a vector needs: `np.array([1, 2, 3]) + np.array([4, 5, 6])` gives `[5, 7, 9]`. Reaching for a list where an array belongs is one of the most common sources of silent, wrong-shaped bugs in early numpy code.

A **matrix** is a two-dimensional array of numbers. An entire dataset is a matrix: N rows (observations) by M columns (features), written $X \in \mathbb{R}^{N \times M}$.

The **dot product** of two vectors, $\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i$, is the sum of their element-wise products. It is the fundamental operation behind prediction: a linear model's output is the dot product between a feature vector and a weight vector. Concretely: if $\mathbf{a} = [2, 3]$ and $\mathbf{b} = [4, 1]$, then $\mathbf{a} \cdot \mathbf{b} = (2 \times 4) + (3 \times 1) = 8 + 3 = 11$. Multiply each pair of matching elements, then add the results. A tiny prediction looks the same. A customer with 5 months' tenure and \$80 monthly spend is the feature vector $\mathbf{x} = [5, 80]$. A model's learned weights might be $\mathbf{w} = [0.1, 0.05]$. The prediction is $\mathbf{x} \cdot \mathbf{w} = (5 \times 0.1) + (80 \times 0.05) = 0.5 + 4 = 4.5$. Every linear model prediction, no matter how many features, is this same operation done at larger scale.

Matrix multiplication generalises the dot product across many vectors at once (if X is $N \times M$ and W is $M \times K$, then XW is $N \times K$). Concretely: let $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$. Each entry of AB is a dot product: row i of A against column j of B. The top-left entry is row 1 of A, $[1, 2]$, dotted with column 1 of B, $[5, 7]$: $(1 \times 5) + (2 \times 7) = 19$. Working through all four entries the same way gives

$$AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Nothing new happened here. It's the same multiply-then-add from the dot product example above, done four times, once per output entry. This single operation, chained repeatedly with non-linearities in between, is how every neural network layer transforms its input.

The **transpose** of a matrix, $(X^\top)_{ij} = X_{ji}$, flips rows and columns. It appears throughout ML's closed-form solutions.

What Is the Identity Matrix?

The **identity matrix** I is the matrix equivalent of the number 1: square, with 1s down the main diagonal and 0s everywhere else, for example $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ in two dimensions. Multiplying anything by it leaves that thing unchanged, $AI = A$, the same way multiplying a number by 1 leaves it unchanged.

The **inverse** A^{-1} of a square matrix satisfies $AA^{-1} = I$: the matrix version of "a number times its reciprocal equals 1." It exists only for square, non-singular matrices. In practice, computing A^{-1} directly is both slower and numerically less stable than solving a linear system with `np.linalg.solve`. Prefer `np.linalg.solve` whenever the goal is to solve $Ax = b$, rather than to obtain A^{-1} as an object.

This explains several things that would otherwise be opaque rules of thumb. Feature scaling matters because dot products and distance metrics are sensitive to the raw magnitude of each feature. A feature measured in the tens of thousands (income) will dominate a distance calculation against a feature measured in single digits (age) unless both are normalised onto comparable scales. Principal Component Analysis, covered in a later chapter, works by finding the directions of maximum variance in the data matrix. That computation is, at its core, an eigendecomposition of the data's covariance matrix.

And neural networks can approximate complex, highly non-linear functions precisely because they interleave linear (matrix multiplication) and non-linear (activation function) steps. The linear algebra provides the structure, the non-linearities provide the expressive power.

What Is an Eigendecomposition?

Multiplying a vector by a matrix usually changes both its direction and its length. An **eigenvector** of a matrix is a special vector that keeps its direction. The matrix only stretches or shrinks it, by a factor called the **eigenvalue**: $Av = \lambda v$. A simple case makes this concrete. Let $A = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$. Applying A to $[1, 0]$ gives $[2, 0]$, which is just $[1, 0]$ scaled by 2. Applying A to $[0, 1]$ gives $[0, 3]$, scaled by 3. Those two directions, and their scaling factors 2 and 3, are A's eigenvectors and eigenvalues. An **eigendecomposition** finds all of a matrix's eigenvectors and eigenvalues at once, breaking the matrix down into its fundamental stretch directions. This is exactly what PCA uses. The eigenvector with the largest eigenvalue points in the direction the data is most spread out in. That's the definition of a principal component.

3.2.5 Linear Regression as a Matrix Equation

The simplest supervised model, linear regression, is where probability, statistics, and linear algebra meet. In matrix form, predictions are

$$\hat{y} = Xw$$

where X is the $N \times M$ feature matrix, w is the $M \times 1$ weight vector, and $\hat{y}$ is the resulting $N \times 1$ vector of predictions. "Learning" here means finding the w that minimises the sum of squared errors between $\hat{y}$ and the true y (a statistics objective). That minimisation has an exact, closed-form solution called the **normal equations**:

$$w = (X^\top X)^{-1} X^\top y$$

This is not an approximation. It's not an iterative procedure. It is the exact minimiser, derivable in a few lines of calculus (Chapter 4 covers the calculus side properly). It is the same answer `sklearn.linear_model.LinearRegression` returns. sklearn gets there with a more numerically stable least-squares solver rather than forming $X^\top X$, but the minimiser is the same. The Worked Example below verifies this by hand against the `iris` dataset and confirms the numpy and sklearn results agree to six decimal places.

3.3 Trade-offs and Limits

The depth of maths needed scales with the job at hand. Anywhere model uncertainty, calibration, or Bayesian methods are directly in play, the probability theory in this chapter is required. Murphy's introductory text goes considerably further into formal probability than this chapter attempts. But for the large majority of day-to-day ML work, sklearn, XGBoost, and similar libraries do the underlying matrix algebra and optimisation internally. The practitioner's job is not to re-derive the normal equations from scratch every time. The job is to have enough intuition that nothing the library does is surprising, and enough fluency to debug when results look wrong. A model that suddenly produces `NaN` predictions after a feature-scaling change, for instance, is far easier to diagnose with an intuition for numerical instability in matrix inversion than without one.

The p-value trap deserves its own note here because it recurs throughout applied statistics. Statistical significance is not the same claim as practical importance. Treating $p < 0.05$ as a stopping condition (rather than one input alongside effect size and business context) is one of the most common statistical mistakes in industry data science, not just academic research.

3.4 Procedures

Procedure 3.1 (Compute and Interpret Correlation). *When to use:* during exploratory data analysis, to identify which features relate to the target and to each other.

Listing 3.1 Computing a correlation matrix and ranking features by correlation with the target

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

r = df['feature_a'].corr(df['target'])
print(f"r = {r:.3f}")

corr = df.corr(numeric_only=True)

sns.heatmap(corr, annot=True, fmt='.2f', cmap='RdBu_r',
            center=0, vmin=-1, vmax=1, square=True)
plt.tight_layout()
plt.show()

corr_with_target = (
    corr['target']
    .abs()
    .sort_values(ascending=False)
)
print(corr_with_target.head(10))
```

Common mistake: correlation measures *linear* relationships only. A feature can carry strong non-linear predictive power while its Pearson r sits near zero. Always inspect the scatter plot, not just the coefficient.

Procedure 3.2 (Test for Normality). *When to use:* before applying any statistical test that assumes Gaussian data: the t-test, ANOVA, or linear regression residual diagnostics.

Listing 3.2 Visual and statistical normality checks, in the order they should be applied

```
from scipy import stats
import matplotlib.pyplot as plt

data = df['column_name'].dropna()

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
axes[0].hist(data, bins=30)
axes[0].set_title('Histogram')
stats.probplot(data, plot=axes[1])
axes[1].set_title('Q-Q Plot (straight line = Gaussian)')
plt.tight_layout()
plt.show()

stat, p = stats.shapiro(data)          # best for n < 5000
print(f"Shapiro-Wilk: W={stat:.4f}, p={p:.4f}")
print("Gaussian" if p > 0.05 else "Not Gaussian")

stat, p = stats.normaltest(data)       # better for larger samples
print(f"D'Agostino-Pearson: p={p:.4f}")
```

If the data isn't Gaussian, prefer median/IQR over mean/std for summaries. Use the Mann-Whitney U test instead of a t-test. Consider log-transforming a right-skewed variable with `np.log1p`.

Procedure 3.3 (Perform Core Matrix Operations in numpy). *When to use:* any time feature matrices need to be manipulated directly, or an ML operation needs to be implemented from first principles.

Listing 3.3 The numpy operations that cover almost every matrix manipulation an ML practitioner needs

```
import numpy as np

A = np.array([[1, 2], [3, 4], [5, 6]])  # shape (3, 2)
B = np.array([[7, 8], [9, 10]])          # shape (2, 2)

A.T                # transpose, shape (2, 3)

# matrix multiply - ALWAYS use @, not np.dot, for 2D arrays
C = A @ B
```

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
v1 @ v2            # dot product = 1*4 + 2*5 + 3*6 = 32

M = np.array([[2.0, 1.0], [1.0, 3.0]])

# equivalent to M^{-1}, computed more stably
np.linalg.solve(M, np.eye(2))

# solves M @ x = b directly - prefer this over inv()
b = np.array([1.0, 2.0])
np.linalg.solve(M, b)

eigenvalues, eigenvectors = np.linalg.eig(M)   # used in PCA

np.linalg.norm(v1)               # L2 norm (Euclidean length)
np.linalg.norm(v1, ord=1)        # L1 norm (Manhattan distance)

X = np.random.randn(100, 5)      # (N, M) data matrix
w = np.random.randn(5)           # (M,) weight vector
y_hat = X @ w                    # (N,) predictions
```

Common mistake: `M * M` is element-wise multiplication, not matrix multiplication (`M @ M`). Mixing up `*` and `@` on two matrices of the same shape produces a result that runs without error but is mathematically wrong.

Procedure 3.4 (Interpret a p-value Correctly). *When to use:* whenever reporting a statistical test result or evaluating a claim about model performance.

Listing 3.4 A two-sample t-test paired with its effect size, as they should always be reported together

```
from scipy import stats
import numpy as np

group_a = df[df['group'] == 'A']['metric']
group_b = df[df['group'] == 'B']['metric']
t_stat, p_value = stats.ttest_ind(group_a, group_b)
print(f"t = {t_stat:.3f}, p = {p_value:.4f}")

# simple pooled std: fine when the groups are the same size
pooled_std = np.sqrt((group_a.std()**2 + group_b.std()**2) / 2)
cohens_d = (group_a.mean() - group_b.mean()) / pooled_std
# 0.2=small, 0.5=medium, 0.8=large
print(f"Cohen's d = {cohens_d:.3f}")
```

Common mistake: treating $p > 0.05$ as proof of "no effect." It only means "insufficient evidence of an effect," which is a materially weaker claim. A related mistake is running many hypothesis tests and reporting only the ones that happen to clear $p < 0.05$ (p-hacking). When testing K hypotheses at once, apply a Bonferroni correction and use threshold $p < 0.05/K$ instead.

Procedure 3.5 (Compute Summary Statistics by Group). *When to use:* understanding how a feature differs across classes, a standard step in exploratory data analysis.

Listing 3.5 Grouped statistics and a simple z-score outlier check

```
(
    df.groupby('species')[numeric_cols]
    .agg(['mean', 'std', 'median'])
    .round(3)
)

# check for heteroscedasticity
df.groupby('species')['feature'].var()

df[numeric_cols].cov()

z_scores = (df[col] - df[col].mean()) / df[col].std()
outliers = df[z_scores.abs() > 3]
print(f"{len(outliers)} outliers in {col}")
```

3.5 Worked Example

The companion notebook builds a complete mathematical toolkit in three parts, each verifying a claim from the Exposition against real computation rather than leaving it as assertion.

Part 1 — the Law of Large Numbers and Bayes' Theorem. Simulating 10,000 fair coin flips (`np.random.binomial(1, p=0.5, size=10_000)`, seed 42) and tracking the running frequency of heads shows convergence towards the true probability, exactly as claimed. The observed frequency after all 10,000 flips is **0.4924**, visibly closer to 0.5 than the frequency after only a handful of flips. The gap shrinks roughly as $1/\sqrt{n}$.

Listing 3.6 The disease-test Bayes' theorem calculation, computed rather than asserted

```
P_disease = 0.01
P_pos_given_dis = 0.99
P_neg_given_healthy = 0.95
P_pos_given_healthy = 1 - P_neg_given_healthy
P_healthy = 1 - P_disease

P_positive = (
    (P_pos_given_dis * P_disease)
    + (P_pos_given_healthy * P_healthy)
)
P_disease_given_pos = (P_pos_given_dis * P_disease) / P_positive

print(f"P(positive):                {P_positive:.3f}")
print(f"P(disease | test positive): {P_disease_given_pos:.1%}")
```

Running this gives `P(positive) = 0.059` and `P(disease | test positive) = 16.7%`. That matches the Exposition's hand-worked figure exactly. It's confirmed independently by a direct simulation over 100,000 synthetic people, counting true and false positives directly rather than trusting the algebra alone.

Part 2 — full statistical analysis of the iris dataset. The classic four-feature, three-species iris dataset (150 rows) makes mean-vs-median skew concrete. `petal length (cm)` has mean 3.758 cm against median 4.350 cm, a difference of 0.592 cm. That's by far the largest gap of the four features (the next largest, `petal width`, differs by only 0.101 cm). The reason: the dataset mixes three species whose petal lengths cluster in different ranges, producing a bimodal, not symmetric, overall distribution. The correlation matrix confirms the strongest linear relationship in the dataset is between `petal length` and `petal width`, at $r = 0.963$. That's visibly higher than `sepal length` vs. `petal length` ($r = 0.872$) or `sepal length` vs. `petal width` ($r = 0.818$). Running Shapiro-Wilk normality tests (Procedure 3.2) on each feature across all species combined rejects Gaussianity for `sepal length` ($p = 0.010$), `petal length` ($p = 7.4 \times 10^{-10}$) and `petal width` ($p = 1.7 \times 10^{-8}$). That's unsurprising, since mixing three species' distributions produces multimodality. Restricting to `setosa` alone brings both `sepal length` ($p = 0.46$) and `sepal width` ($p = 0.27$) comfortably back into "consistent with Gaussian" territory. This illustrates that Shapiro-Wilk tests the *sample actually given to it*, not some platonic notion of the feature.

Part 3 — linear regression from scratch, verified against sklearn. Using `petal length` to predict `petal width` (Listing 3.7), solving the normal equations directly with `np.linalg.solve` produces:

Listing 3.7 Solving the normal equations directly and cross-checking against sklearn

```
X_raw = df[['petal length (cm)']].values
y = df['petal width (cm)'].values
# add bias column
X = np.column_stack([np.ones(len(X_raw)), X_raw])

XtX = X.T @ X
Xty = X.T @ y
w = np.linalg.solve(XtX, Xty)
print(f"Intercept (w0): {w[0]:.4f}")
print(f"Slope     (w1): {w[1]:.4f}")
```

`w = [-0.3631, 0.4158]`, i.e. `petal_width ~ -0.3631 + 0.4158 × petal_length`. Fitting `sklearn.linear_model.LinearRegression` on the same data gives intercept `-0.363076` and coefficient `0.415755`. That matches the numpy result to six decimal places, confirming these are the same computation under two different names. The fitted model achieves $R^2 = 0.9271$ and RMSE $= 0.2051$ cm. Petal length alone explains close to 93% of the variance in petal width across all three species, with an average prediction error of about two millimetres. That's a strikingly tight fit for a single-feature linear model. It's a direct consequence of the $r = 0.963$ correlation already observed in Part 2: for simple linear regression, R^2 is exactly r^2 ($0.963^2 = 0.927$).

3.6 AWS SageMaker Notes

AWS SageMaker Equivalent

```
# SageMaker Data Wrangler provides a UI for
# statistical analysis at scale. For programmatic
# stats on datasets too large for local memory, use
# SageMaker Processing Jobs instead:

import sagemaker
from sagemaker.processing import ScriptProcessor

# processor = ScriptProcessor(
#     image_uri=...,
#     role=role,
#     instance_type='ml.m5.xlarge',
#     instance_count=1,
# )
# processor.run(code='stats_script.py', ...)
```

> **When to prefer SageMaker Processing:** once a dataset no longer fits comfortably in local memory (roughly beyond 10 GB). Below that threshold, pandas and scipy running locally are faster to iterate with and cost nothing per run. Reach for managed infrastructure only once local computation is the bottleneck.

3.7 History and Further Reading

Murphy's introductory volume gives the formal treatment this chapter deliberately keeps informal. Chapter 2 covers probability theory properly (joint, marginal, and conditional distributions). Chapter 3 covers estimation, hypothesis testing, and confidence intervals in full (Murphy 2022). The same book's Appendix B is, in the author's own words, exactly the depth of linear algebra review an ML practitioner needs, no more, no less (Murphy 2022). Burkov's opening chapter is worth a quick pass for its notation conventions and its concise statement of the supervised learning setup. This chapter's linear regression example builds on that setup directly (Burkov 2019, 1–10). This chapter keeps the linear algebra informal. For the fully rigorous treatment, Humpherys, Jarvis, and Evans' *Foundations of Applied Mathematics, Volume 1: Mathematical Analysis* builds vector spaces axiomatically over arbitrary fields, rather than this chapter's concrete $\mathbb{R}^n$ starting point. It's worth a look for exactly that contrast (Humpherys et al. 2017). Chu-Carroll's *Good Math* is a different kind of companion. It's not a probability or linear algebra text, but an accessible, narrative tour of number systems, logic, and computability. It gives the curious reader a broader sense of the mathematical foundations sitting underneath this chapter's notation (Chu-Carroll 2013). Beyond formal texts, 3Blue1Brown's "Essence of Linear Algebra" video series is widely regarded as the best available visual intuition for matrix operations. It gives geometric intuition for transposes, matrix multiplication as a transformation, and eigenvectors, in a way no amount of algebraic manipulation alone conveys.

3.8 How This Connects

This chapter builds on Chapter 2. With a defined ML problem in hand, this chapter supplies the mathematical language needed to describe precisely what that problem is optimising. It enables Chapter 4 (Calculus & Optimisation), which picks up exactly where the normal equations' limitations leave off: most real models have no closed-form solution and need gradient descent instead. It also enables the supervised learning chapters later in this book, and the neural network chapters in Book 2. It's often confused with a demand for mathematical *expertise*. What it requires is mathematical *literacy*: the ability to read an equation, recognise which operation it describes, and reason about what changing an input does to the output, not the ability to derive every result from a blank page.

3.9 Exercises

1. [**W**] Using the disease-test numbers from Listing 3.6, explain in your own words why $P(\text{disease} \mid \text{positive})$ comes out so much lower than the test's 99% sensitivity would naively suggest.
2. [**W**] For each of the following, name the distribution (Gaussian, Bernoulli, Binomial, or Poisson) that best describes it, and justify your choice in one sentence: (a) whether a single email is spam; (b) the number of fraudulent transactions in a given hour; (c) the number of heads in 50 coin flips; (d) the measurement error of a sensor reading a stable physical quantity.
3. [**A**] Using Procedure 3.1 and the iris dataset, compute the correlation between `sepal width (cm)` and every other feature. Which pair has the weakest (closest to zero) correlation, and does that match the intuition that sepal width is the "odd one out" of the four iris measurements?
4. [**A**] Extend Listing 3.7 to use two features, `petal length (cm)` and `sepal length (cm)`, to predict `petal width (cm)`. You only need to change how `X_raw` is constructed. The normal-equations solve is unchanged. Report the new R^2 and state whether adding the second feature meaningfully improved the fit over the single-feature model.
5. [**A**] Using Listing 3.4, run a two-sample t-test comparing `petal length (cm)` between `setosa` and `versicolor` in the iris dataset. Report the p-value and Cohen's d, and state in one sentence whether the difference is both statistically significant and practically large.
6. [**E**] A colleague reports "we found a statistically significant improvement, $p = 0.002$" as the sole justification for shipping a model change. Using this chapter's discussion of p-values and effect sizes, write the two or three follow-up questions you'd ask before agreeing the change should ship.
7. [**E**] Section "Linear Algebra" argues that feature scaling matters because dot products and distance metrics are sensitive to raw magnitude. Construct a small numeric example (two features, three or four data points) where an unscaled Euclidean distance calculation is dominated by one feature, and show the same calculation after standardising both features to mean 0 and standard deviation 1.
8. [**R**] Read Murphy's Appendix B on linear algebra (Murphy 2022). Find one operation or concept covered there that this chapter didn't mention, and explain in a short paragraph where it would become relevant later in this book (for instance, in the PCA or neural network chapters).

3.10 Mental Model

A dataset is a matrix. A model is a function that transforms that matrix into predictions. A training objective is an equation that measures how wrong those predictions are. Probability is the language that objective is usually expressed in. Statistics is how confidence in a result, or lack of it, gets quantified once the model exists. None of this requires being a mathematician. It requires being able to read these equations without flinching. From here on, every chapter in this book assumes exactly that.

4 Maths Foundations: Calculus & Optimisation

Chapter 3 gave linear regression a closed-form solution: the normal equations solve for the exact optimum in one step. The underlying loss surface is a simple quadratic bowl, with a single minimum reachable by matrix algebra. That trick doesn't generalise. Logistic regression, neural networks, and nearly every other model this book covers have no closed-form solution. They have to be found by search, starting from a random guess and iteratively improving it until the loss is small enough. This chapter is about that search. What a gradient is. Why walking downhill on a loss surface finds good parameters. Why step size matters so much. Why the chain rule of calculus, applied billions of times per training run, is the entire mechanism behind training a neural network. Everything from logistic regression in Chapter 11 through fine-tuning language models later in this series runs on the ideas in this chapter.

4.1 Motivation

Training a model is optimisation: find the parameters that minimise a loss function measuring how wrong the model's predictions are. The loss function defines a surface over parameter space. For a model with two weights, imagine literal terrain, with height at each point equal to the loss for those two weight values.

Training is the search for the lowest point in that terrain. Gradient descent is the specific search algorithm this book relies on almost everywhere: stand where you are, work out which direction goes downhill steepest, take a step, repeat. It scales from a single-parameter toy problem to models with hundreds of billions of parameters. That's why it, not exact algebraic solutions, is the workhorse of modern machine learning.

4.2 Exposition

4.2.1 Derivatives and Gradients

A **derivative** measures a function's rate of change at a point. You already have an intuition for this from somewhere far more familiar than calculus class. Velocity is the derivative of

position with respect to time: how fast something is moving, right now. Acceleration is the derivative of velocity with respect to time: how fast the speed itself is changing. A car's speedometer is displaying a derivative in real time, and pressing the accelerator changes a derivative of a derivative. For a function $f(x)$, the derivative $\frac{df}{dx}$ evaluated at x asks the same question in general form: if x increases by a tiny amount, how much does f change, and in which direction? The sign carries the information that matters for optimisation. If $\frac{df}{dx} > 0$, f is increasing at x, so decreasing x reduces f. If $\frac{df}{dx} < 0$, the reverse. If $\frac{df}{dx} = 0$, the function is momentarily flat, a candidate minimum, maximum, or saddle point.

Real loss functions depend on many parameters at once: one weight per feature, easily thousands in a modest model. The multivariate generalisation of the derivative is the **gradient**, ∇f, the vector of partial derivatives, one per parameter:

$$\nabla f(\mathbf{w}) = \left[\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \dots, \frac{\partial f}{\partial w_n} \right]$$

Each entry says how the loss responds to nudging that one parameter while holding the others fixed. The gradient vector as a whole points in the direction of **steepest ascent** of f. To reduce the loss, step in the *opposite* direction. That's the basis of gradient descent.

4.2.2 Gradient Descent

The update rule follows directly from the previous section:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \, \nabla f(\mathbf{w})$$

where $\mathbf{w}$ is the current parameter vector, α (the **learning rate**) controls how large a step to take, and $\nabla f(\mathbf{w})$ is the gradient of the loss at the current parameters. Repeating this update (recompute the gradient, take a step, recompute, take another step) is gradient descent. It ends, in practice, when the loss stops decreasing meaningfully (convergence), or after a fixed number of iterations.

Nothing in this rule requires knowing the shape of the whole loss surface in advance. Gradient descent only ever needs the *local* slope at the current position. That's exactly why it scales to problems too large to visualise or solve algebraically.

4.2.3 The Learning Rate

The learning rate α governs a trade-off in every iterative optimiser, and there's no universal right answer for it.

Too large, and each step overshoots the minimum. The parameters bounce past the bottom of the bowl, land higher up the opposite wall. The loss oscillates or diverges outright, getting *worse* with each iteration instead of better. **Too small**, and steps barely move the parameters. Training still converges eventually. But it may take orders of magnitude more iterations, and so far more compute and wall-clock time, to get there. **Just right** produces steady, roughly monotonic loss decrease that levels off near a minimum.

> **i** What Does Monotonic Mean?
>
> A sequence is **monotonic** if it only moves in one direction: always decreasing, or always increasing, never doubling back. A "roughly monotonic" loss curve mostly falls, step after step, without bouncing up, then down, then up again along the way. It doesn't have to fall by the same amount every step, just not reverse direction for long.

Because the correct value depends on the specific problem, the architecture, and the scale of the input data, practitioners rarely fix a single learning rate for an entire run. Learning rate *schedules*, which reduce α over the course of training, are standard practice because "just right" tends to shrink as training approaches convergence.

4.2.4 The Chain Rule and Backpropagation

Neural networks are compositions of functions. The output of one layer is the input to the next. Computing the gradient of the loss with respect to a weight buried several layers deep requires the **chain rule** from calculus:

> **i** The Chain Rule, in Plain Words
>
> Neural network layers are functions stacked inside each other: the loss depends on the last layer's output, which depends on the layer before that, and so on, all the way back to the first weight. The chain rule is just the rule for how a nudge ripples through a chain like that. Picture a set of gears. Turn the first gear, and it turns the second one at some ratio. The second turns the third at another ratio. How fast the last gear spins, for one turn of the first, is just those ratios multiplied straight through the chain. A derivative is a local ratio like that: how much the output moves for a small nudge to the input. Chaining functions together, $f(g(x))$, chains their local ratios together the same way the gears do. That's the whole content of "the derivative of $f(g(x))$

with respect to x." g is the inner gear, f is the outer one. Their local rates of change multiply to give the rate of change of the whole chain.

$$\frac{d}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)$$

Applied repeatedly, layer by layer, from the loss backward to the earliest weights, this *is* backpropagation. It propagates the gradient of the loss backward through a network, multiplying by each layer's local derivative along the way. Frameworks like PyTorch and TensorFlow automate this via autograd, tracking every operation done on a tensor and replaying the chain rule automatically to compute exact gradients. But the chain rule is the mechanism underneath the automation. Understanding it is what turns autograd from magic into an implementation detail.

4.2.5 Loss Functions

A loss function maps predictions and true labels to a single scalar: the number gradient descent is trying to minimise. The choice of loss function defines what "good" means for a given model. This is a modelling decision to be made as you design your solution.

Mean Squared Error (MSE), for regression:

$$L = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Squaring penalises large errors disproportionately and produces a smooth function that is differentiable everywhere, convenient for gradient-based optimisation.

Binary cross-entropy, for binary classification:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

This isn't arbitrary. The ingredients aren't new. It falls directly out of maximum likelihood estimation under the Bernoulli distribution from Chapter 3. It is zero for a perfect prediction and grows without bound as a confident prediction diverges from the truth.

Mean Absolute Error (MAE), for regression that needs to be robust to outliers:

$$L = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$$

MAE weighs every error linearly rather than quadratically. A handful of extreme outliers can't dominate the loss the way they can under MSE. But it is not differentiable at zero, which forces optimisers to fall back on subgradients there.

4.2.6 Variants of Gradient Descent

The update rule above leaves an open question: the gradient of the loss over *how much* data, computed at each step? Three answers are used in practice, trading off gradient accuracy against speed.

Batch gradient descent computes the gradient over the entire training set before taking a single step. This is the most accurate gradient available. But for large datasets, one step becomes prohibitively slow to compute.

Stochastic gradient descent (SGD) computes the gradient from a single example at a time. Each individual step is nearly free to compute, but noisy. An update based on one example may not point in the true downhill direction at all. Counterintuitively, that noise isn't always negative: it can help the optimiser escape shallow local minima that a smoother trajectory would settle into.

Mini-batch gradient descent computes the gradient over a small batch of examples, typically 32, 64, 128, or 256 at a time. This is the practical compromise used throughout neural network training. Batches are large enough to average out most of the single-example noise. Small enough to keep each step fast. And a natural fit for GPU parallelism.

4.2.7 Momentum and Adam

Plain gradient descent treats every step independently, with no memory of where it has already been. That makes it slow to cross flat regions, and prone to zig-zagging in narrow, curved valleys. Two refinements address this.

Momentum accumulates a running velocity that blends the new gradient with the previous step's direction:

$$\mathbf{v} \leftarrow \beta\mathbf{v} + (1-\beta)\nabla f(\mathbf{w}), \qquad \mathbf{w} \leftarrow \mathbf{w} - \alpha\mathbf{v}$$

Consistent gradient directions reinforce each other and accelerate. Oscillating directions partially cancel and dampen. The effect is qualitatively like a ball rolling downhill, carrying momentum through small bumps rather than reacting to every local wiggle.

Adam (Adaptive Moment Estimation) goes further, combining momentum with a *per-parameter* adaptive learning rate. It tracks both a running average of the gradient (like momentum) and a running average of the squared gradient. Then it scales each parameter's

step inversely with the typical magnitude of that parameter's recent gradients. Parameters with consistently large gradients get smaller effective steps. Parameters with small gradients get larger ones. The practical consequence is that Adam is less sensitive to the initial choice of learning rate than SGD. That's why it is the default optimiser for training neural networks.

4.2.8 When scipy Can Help, and When It Can't

For classical, low-dimensional optimisation problems (fitting a curve, tuning a handful of hyperparameters, minimising a smooth well-behaved function), `scipy.optimize.minimize` provides solvers that are faster and more reliable than a hand-rolled gradient descent loop. It is the right tool for hyperparameter search, fitting statistical models, and any smooth optimisation that can use the full dataset at every step. Its L-BFGS-B solver handles millions of parameters on such problems. (`minimize` defaults to BFGS when there are no bounds, and to L-BFGS-B when there are.)

It is not a substitute for mini-batch gradient descent when training a neural network, for three reasons. First, L-BFGS-B needs exact, low-noise gradients. Its line search and its curvature estimates are built on full-batch gradients, and mini-batch noise breaks both. Second, mini-batch stochasticity is not noise we want to eliminate. It is how SGD and Adam escape saddle points that riddle non-convex loss surfaces. Third, realistic training data rarely fits in memory all at once. Batch-by-batch processing is required regardless of which optimiser is used. Memory for the solver itself is not the obstacle. The "L" in L-BFGS stands for limited-memory: it keeps only the last few gradient differences, typically about 10 pairs, so its memory grows linearly with the parameter count. The square-size cost belongs to exact Newton's method, which stores the full Hessian.

4.3 Trade-offs and Limits

Convex versus non-convex loss surfaces. For a convex loss function (logistic regression's is one example), gradient descent with a small enough learning rate is guaranteed to reach the global minimum. A convex bowl has no other minima to get stuck in. Too large a learning rate still diverges, as the Worked Example shows. Neural network loss surfaces are not convex. They are riddled with local minima and saddle points. Gradient descent finds *a* good minimum, not provably *the* best one.

Recall the terrain from this chapter's Motivation: for a model with two weights, height at each point is loss. A convex loss function is that terrain shaped like a single bowl, one valley, one lowest point, no way to get stuck anywhere else. A non-convex loss function is terrain like this instead:

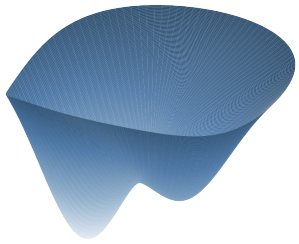


Figure 4.1: A non-convex loss surface: a deep valley (the global minimum) and a shallower one beside it (a local minimum), separated by a ridge.

Gradient descent dropped into the shallow valley on the right has no way to know the deeper one exists. Every direction from its floor goes up, so it stops, converged, and wrong about being done. That is what "riddled with local minima and saddle points" looks like on real terrain, not just as a phrase.

In practice, for large neural networks, the local minima found this way tend to generalise well regardless. This is an empirical observation about large models. Not a theorem.

First-order versus second-order methods. Gradient descent uses only first derivatives, the slope. Newton's method and its relatives also use second derivatives: curvature, captured in the Hessian matrix. That lets them converge in far fewer iterations near a minimum. The catch is that the Hessian is an $M \times M$ matrix for M parameters, so storing and inverting it becomes infeasible long before M reaches the billions-to-trillions scale of modern neural networks. Limited-memory quasi-Newton methods such as L-BFGS avoid storing it, but they need exact, full-batch gradients, which neural network training doesn't have. This is why first-order adaptive methods like Adam, not second-order methods, dominate deep learning. The trade-off is convergence speed per step, against whether the method can run at all. At GenAI scale that's barely a contest: second-order methods can't run at that size to begin with.

4.4 Procedures

Procedure 4.1 (Choose a Starting Learning Rate). *When to use:* at the start of any training run, before the first real epoch.

Listing 4.1 A short learning-rate range test, and rule-of-thumb starting points

```
import numpy as np

def lr_range_test(X, y, lr_min=1e-5, lr_max=1.0, n_trials=30):
    """Quick scan to find a reasonable learning-rate range."""
    lrs = np.logspace(np.log10(lr_min), np.log10(lr_max), n_trials)
    final_losses = []
    for lr in lrs:
        w = np.zeros(X.shape[1])
        for _ in range(50):  # short run
            w = w - lr * mse_gradient(w, X, y)
        final_losses.append(mse_loss(w, X, y))
    best_lr = lrs[np.argmin(final_losses)]
    return best_lr, lrs, final_losses

# Rule-of-thumb starting points:
#   simple linear/logistic models:  lr = 0.1
#   neural networks (Adam default): lr = 1e-3
#   fine-tuning large models:       lr = 1e-5 to 5e-5
#   XGBoost:                        lr = 0.05 to 0.3
```

Common mistake: picking a learning rate once and never revisiting it. A rate that is "just right" early in training is frequently too large once the loss has mostly converged, which is the problem learning rate schedules (Procedure 4.4) exist to solve.

Procedure 4.2 (Read a Loss Curve). *When to use:* after every training run, as the first diagnostic before touching any hyperparameter.

Listing 4.2 Plotting train/validation loss and the diagnostic table behind it

```
import matplotlib.pyplot as plt

def plot_loss_curve(
    train_losses, val_losses=None, title='Training Loss'
):
    plt.figure(figsize=(10, 4))
    plt.plot(
        train_losses,
        label='Train loss', color='#4E79A7', linewidth=2,
    )
    if val_losses is not None:
        plt.plot(
            val_losses,
```

```
            label='Validation loss',
            color='#E15759', linewidth=2,
        )
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    plt.show()

patterns = {
    "Steadily decreasing": "Good: model is learning",
    "Decreasing then plateau": (
        "Good: converged. More epochs won't help"
    ),
    "Oscillating": "Learning rate too high: reduce by 3-10x",
    "Train low, val high": (
        "Overfitting: add regularisation or more data"
    ),
    "Both high, not decreasing": (
        "Underfitting: model too simple, or LR too low"
    ),
    "NaN or infinity": (
        "Learning rate too high, or a numerical issue"
    ),
}
```

Procedure 4.3 (Use scipy.optimize for Classical Optimisation). *When to use:* smooth optimisation that can use the full dataset at every step, curve fitting, or whenever a precise minimum matters more than raw training speed.

Listing 4.3 scipy.optimize.minimize with and without a supplied gradient

```
import scipy.optimize as opt
import numpy as np

def my_loss(params):
    w0, w1 = params
    predictions = w0 + w1 * X
    return np.mean((predictions - y) ** 2)

def my_gradient(params):
    w0, w1 = params
    residuals = (w0 + w1 * X) - y
    return np.array([
```

```
        2 * np.mean(residuals),
        2 * np.mean(residuals * X),
    ])

result = opt.minimize(
    fun=my_loss, x0=[0.0, 0.0],
    jac=my_gradient,     # supplying the gradient converges faster
    method='L-BFGS-B', options={'maxiter': 1000, 'ftol': 1e-9}
)
print(result.success, result.x, result.fun)
```

Common mistake: reaching for `scipy.optimize.minimize` on a neural network because "it converges faster" on small problems. See this chapter's Exposition on why scipy's solvers don't suit neural network training. Exact Newton methods can't store the Hessian past tens of thousands of parameters. L-BFGS-B can run at that size, but it needs exact, full-batch gradients.

Procedure 4.4 (Schedule a Decaying Learning Rate). *When to use:* training runs long enough that a single fixed learning rate is a poor fit across the whole run.

Listing 4.4 Step decay, cosine decay, and warmup-then-cosine schedules

```
import numpy as np

def step_decay(initial_lr, epoch, drop=0.5, epochs_drop=20):
    return initial_lr * (drop ** (epoch // epochs_drop))

def cosine_decay(initial_lr, epoch, total_epochs):
    return initial_lr * 0.5 * (
        1 + np.cos(np.pi * epoch / total_epochs)
    )

def warmup_cosine(initial_lr, epoch, warmup_epochs, total_epochs):
    if epoch < warmup_epochs:
        # epoch + 1, so the first epoch already makes an update
        return initial_lr * (epoch + 1) / warmup_epochs
    return cosine_decay(
        initial_lr,
        epoch - warmup_epochs,
        total_epochs - warmup_epochs,
    )
```

Procedure 4.5 (Spot Vanishing or Exploding Gradients). *When to use:* a neural network's loss refuses to decrease, or diverges outright, and the learning rate alone doesn't explain it.

Listing 4.5 Checking gradient norms by hand, and the PyTorch equivalent

```
def check_gradient(grad, name='gradient'):
    norm = np.linalg.norm(grad)
    if norm < 1e-7:
        print(f"WARNING: {name} may be vanishing "
              f"(norm={norm:.2e})")
    elif norm > 100:
        print(f"WARNING: {name} may be exploding "
              f"(norm={norm:.2e})")
    else:
        print(f"  {name} norm: {norm:.4f}  (healthy)")
    return norm

# In PyTorch (autograd tracks gradients per parameter,
# automatically):
#   for name, param in model.named_parameters():
#       grad_norm = param.grad.norm().item()
# Standard fix for exploding gradients:
#   torch.nn.utils.clip_grad_norm_(
#       model.parameters(), max_norm=1.0
#   )
```

Common mistake: treating every diverging loss as a learning-rate problem. A loss that goes to NaN after training stably for many epochs is more often exploding gradients deep in the network than a suddenly-wrong learning rate. Check gradient norms (Listing 4.5) before cutting α.

4.5 Worked Example

The companion notebook builds gradient descent up in three stages. It starts on a toy 1D parabola. It moves to a real linear regression problem where the answer can be checked independently. It ends with a comparison across three learning rates, which makes the "too large / too small / just right" trade-off concrete.

Stage 1 — gradient descent on $f(x) = x^2$. With `df(x) = 2*x`, starting at $x = 3.0$ and a learning rate of 0.2, running the update rule for 20 steps:

Listing 4.6 Gradient descent converging on the minimum of f(x) = x²

```
def gradient_descent_1d(start_x, learning_rate, n_steps):
    x = start_x
    history = [(x, f(x))]
    for step in range(n_steps):
        gradient = df(x)
        x = x - learning_rate * gradient
        history.append((x, f(x)))
    return history

history = gradient_descent_1d(
    start_x=3.0, learning_rate=0.2, n_steps=20
)
```

The trajectory shrinks geometrically towards the true minimum at $x = 0$. Step 0 sits at $x = 3.000000$ ($f = 9.0$). By step 3 it has fallen to $x = 0.648000$ ($f = 0.419904$). By step 7, to $x = 0.083981$ ($f = 0.007053$). By step 20, to $x = 0.000110$, with $f(x) \approx 1.2 \times 10^{-8}$. That's visibly geometric decay, since each step multiplies the distance to the minimum by a constant factor of $(1 - 2\alpha) = 0.6$ for this particular f and α.

Stage 2 — training real linear regression by gradient descent, with an independent check. Using a synthetic 200-sample, single-feature regression dataset (`make_regression`, `noise=20`, `random_state=42`), standardised and with a bias column prepended. Full-batch gradient descent is run for 200 epochs at $\alpha = 0.1$:

Listing 4.7 Training linear regression with hand-rolled gradient descent

```
def train_linear_regression_gd(
    X, y, learning_rate=0.1, n_epochs=200
):
    w = np.zeros(X.shape[1])
    loss_history = []
    for epoch in range(n_epochs):
        loss = mse_loss(w, X, y)
        grad = mse_gradient(w, X, y)
        w = w - learning_rate * grad
        loss_history.append(loss)
    return w, loss_history

w_gd, loss_history = train_linear_regression_gd(
    X, y, learning_rate=0.1, n_epochs=200
)
```

The loss falls from 6934.10 at epoch 0 to 406.52 by epoch 40, and stays there. Training has converged before the 200-epoch budget is used up. The real test is whether gradient descent's answer matches the two ways of getting an *exact* answer to the same problem: the normal equations from Chapter 3, and scikit-learn's `LinearRegression`. All three agree to six decimal places: $\mathbf{w} = [-0.689864,\ 80.790445]$ from gradient descent, $[-0.689864,\ 80.790445]$ from the normal equations, and $[-0.689864,\ 80.790445]$ from sklearn. That's an exact match (`np.allclose(..., atol=0.01)` returns `True`). It confirms that iterative search and closed-form algebra are finding the same optimum, as convexity of the MSE loss guarantees they must.

Stage 3 — the learning rate trade-off, made concrete. Running the same training loop for 100 epochs at three learning rates:

Listing 4.8 Three learning rates on the same problem: too small, just right, too large

```
learning_rates = {
    'Too small (0.001)': 0.001,
    'Just right (0.1)': 0.1,
    'Too large (1.5)': 1.5,
}
for label, lr_val in learning_rates.items():
    w = np.zeros(X.shape[1])
    for epoch in range(100):
        loss = mse_loss(w, X, y)
        grad = mse_gradient(w, X, y)
        w = w - lr_val * grad
```

At $\alpha = 0.001$, the loss has only crawled from 6934.10 down to 4797.88 after all 100 epochs. That's nowhere near the optimal 406.52, illustrating that "too small" is not a safety net. It's slow. At $\alpha = 0.1$, the loss is already down to 481.78 by epoch 10 and has essentially converged to 406.53 by epoch 30. At $\alpha = 1.5$, the loss doesn't decrease at all. By epoch 10 it has blown past 10^8 (the notebook clips the plotted value there to keep the chart readable). That's a clean example of divergence from overshooting, not a bug in the implementation.

Finally, the same problem solved with `scipy.optimize.minimize` (Procedure 4.3, L-BFGS-B with the analytic gradient supplied) converges to the identical answer ($\mathbf{w} = [-0.689864,\ 80.790445]$, loss 406.5247) in just 2 iterations. Gradient descent needed about 40 epochs to get there. That gap is the entire argument for Procedure 4.3's "when to use scipy" guidance: for a 2-parameter smooth convex problem, a second-order-aware solver is better. The notebook then makes the scaling argument concrete rather than asserted. Storing the exact Hessian for a modest 1-million-parameter model, as Newton's method does, would need $10^6 \times 10^6 \times 4 = 4 \times 10^{12}$ bytes of float32 storage. That's about 4,000 GB (3,725

GiB), comfortably past what even a high-end accelerator carries in memory. L-BFGS-B avoids that cost by keeping only a few recent gradient differences, so its memory grows linearly with the parameter count. What stops it at neural-network scale is the gradient: mini-batch gradients are noisy, and its line search and curvature estimates need exact ones. That's why this approach is confined to smooth, full-batch problems, and mini-batch gradient descent (or Adam) takes over at neural-network scale.

4.6 AWS SageMaker Notes

AWS SageMaker Equivalent

```
import sagemaker
from sagemaker.estimator import Estimator

estimator = Estimator(
    entry_point='train.py',
    role=role,
    instance_type='ml.p3.2xlarge',   # GPU instance
    hyperparameters={
        'learning_rate': 0.001,
        'epochs': 100,
        'batch_size': 256,
    }
)
estimator.fit({'train': training_data_uri})
```

When to prefer this: SageMaker Training Jobs handle provisioning the GPU instances that gradient descent runs on. You still write the training loop (exactly the update rule from this chapter). SageMaker manages the compute underneath it. A later chapter on hyperparameter tuning replaces hand-set values like `learning_rate` here with SageMaker Automatic Model Tuning. That searches the hyperparameter space via Bayesian optimisation instead.

4.7 History and Further Reading

Murphy's Introduction, Ch. 8, covers optimisation theory, gradient-based methods, and convergence proofs in more formal depth than this chapter attempts (Murphy 2022). Burkov, Ch. 3, walks through gradient descent concisely using logistic regression as the running example. That's a useful second pass on the same material from a different angle (Burkov 2019). Géron's *Hands-On Machine Learning*, Ch. 4, walks through batch, stochastic, and mini-batch gradient descent side by side in code. It's useful for seeing this chapter's update rule run (Géron 2019). Two resources outside the shared bibliography are worth the reader's

time directly. Gabriel Goh's "Why Momentum Really Works" (distill.pub, 2017) gives an illuminating visual account of momentum. 3Blue1Brown's "Backpropagation" video is, among the many available explanations, the clearest visual walkthrough of the chain rule as it plays out inside a neural network.

4.8 How This Connects

This chapter builds directly on Chapter 3's linear algebra and its normal-equations solution to linear regression. This chapter's Worked Example uses that exact solution as the independent check for gradient descent's answer. It enables every model in the rest of this book. Logistic regression, the tree ensembles and neural networks that follow, and the fine-tuning procedures in later volumes all reduce, at some level, to the update rule in this chapter's Exposition. It is often confused with the loss function. But gradient descent is the *how* of minimisation, and the loss function is the *what* being minimised. Chapter 5's cross-entropy and KL divergence supply another important instance of the "what," measured in the specific language of probability distributions.

4.9 Exercises

1. **[W]** Explain in your own words why $\nabla f(\mathbf{w})$ points in the direction of steepest *ascent*, and why gradient descent then steps in the *negative* gradient direction.
2. **[W]** A training run's loss oscillates wildly between epochs 5 and 40, then goes to NaN at epoch 41. Using Listing 4.2's diagnostic table, name the most likely cause and the first thing to try.
3. **[A]** Using Listing 4.6, rerun gradient descent on $f(x) = x^2$ starting at $x = 3.0$ with learning rate $\alpha = 1.1$ instead of 0.2. Report what happens to x over the first 5 steps. Explain it in terms of the $(1-2\alpha)$ contraction factor mentioned in the Worked Example.
4. **[A]** Using Listing 4.7, rerun training with `learning_rate=0.5` instead of `0.1` for 200 epochs, and report the epoch at which the loss first comes within 1% of the optimal loss (406.52), compared to the ~17 epochs it took at $\alpha = 0.1$.
5. **[A]** Using Listing 4.8 as a starting point, add a fourth learning rate of your choosing between 0.1 and 1.5, run it for 100 epochs. Classify it as behaving more like "just right" or "too large," based on where the loss lands by epoch 30.
6. **[E]** The chapter states that Adam's adaptive per-parameter scaling makes it "less sensitive to the initial choice of learning rate than SGD." Propose an experiment (you don't need to run it) that would test this claim directly against plain SGD. State what result would falsify it.

7. [**E**] A colleague proposes using `scipy.optimize.minimize` with `method='L-BFGS-B'` to fine-tune a 50-million-parameter neural network because "it converged in 2 iterations on our small test problem." Using this chapter's argument on why scipy's solvers don't suit neural network training, write the two or three sentences you'd say to redirect them.
8. [**R**] Read Goh's "Why Momentum Really Works" (cited above). Explain, in terms distinct from this chapter's "rolling ball" intuition, why momentum specifically helps in loss surfaces shaped like a narrow curved valley (an ill-conditioned quadratic). Explain how that differs from the plain "escapes local minima via noise" story told about SGD.

4.10 Mental Model

Training a model is hiking downhill in darkness. You can't see the whole landscape. Only which way is downhill from where you're currently standing. Gradient descent is the algorithm for taking a step based on that local slope. The learning rate is how big a step to take. Adam adds a kind of adaptive GPS, scaling each direction's step size based on how steep that particular direction has been recently. The chain rule is what makes it possible to compute "which way is downhill," even when the loss is buried many layers deep inside a composed function. That's exactly what a neural network is.

5 Information Theory Basics

Why does classification use a logarithm in its loss function? Why do decision trees split on "information gain"? Why does a variational autoencoder's loss include a strange penalty term called KL divergence? Why does RLHF constrain a fine-tuned model with something by the same name? All four questions collapse to a single answer: information theory. The mathematics of uncertainty. It gives a precise vocabulary for how much a distribution doesn't know, how expensive a wrong prediction is, and how far apart two probability distributions sit from each other. This chapter builds that vocabulary from a single equation up through four concepts (entropy, cross-entropy, KL divergence, and mutual information) that recur constantly for the rest of this book, usually without ever being named.

5.1 Motivation

Every concept in this chapter comes from one idea: how "surprising" is an event? Flipping a fair coin and getting heads is unsurprising. Flipping an unfair coin that almost always lands heads and getting tails is very surprising. Claude Shannon formalised this in 1948 with the **information content** (or self-information) of an event with probability p:

$$I(p) = -\log_2(p)$$

Low-probability events carry high information content: they're surprising. High-probability events carry low information content: they're expected. Entropy is the average of this quantity over a whole distribution. Cross-entropy and KL divergence compare two distributions using the same log-probability machinery. Mutual information asks how much one variable's distribution shifts when another is known. This is one idea, viewed from four angles. Those four angles happen to be exactly the four things a data scientist needs when training and evaluating models.

5.2 Exposition

5.2.1 Entropy: How Uncertain Is a Distribution?

Entropy, written $H(P)$, is the average information content of a distribution. For a discrete distribution P over K outcomes:

$$H(P) = -\sum_{x} P(x) \log P(x)$$

Entropy measures uncertainty, not correctness. It says nothing about whether a distribution is *right*, only how spread out it is. Bounded by two extremes. At maximum entropy, the distribution is uniform: with K classes, $H(P) = \log(K)$. There's no information favouring any one outcome over another. At minimum entropy, $H(P) = 0$: a point mass, one outcome with probability 1. Zero uncertainty.

This is why decision trees split on entropy (or the closely related Gini impurity, discussed below). At a tree node, entropy measures how mixed the classes are. A good split is one that produces child nodes with lower combined entropy than the parent. Those children are more predictable, closer to a point mass, than the node they came from. The **information gain** of a split is the entropy reduction it gives.

> **i** What Is Gini Impurity?
>
> **Gini impurity** measures the same thing entropy does, how mixed a set of classes is, but with a different formula:
>
> $$\text{Gini}(P) = 1 - \sum_{x} P(x)^2$$
>
> Picture picking one example at random from a node and labelling it with a random guess drawn from the node's own class proportions. Gini impurity is the probability that guess is wrong. A pure node (every example the same class) has Gini impurity 0. Nobody could guess wrong there. A node split evenly across two classes has Gini impurity 0.5, a coin flip. Like entropy, it bottoms out at a point mass and peaks when classes are evenly mixed. The Trade-offs section below compares the two.

5.2.2 Cross-Entropy: The Cost of Being Wrong

Cross-entropy measures how well a predicted distribution Q approximates a true distribution P:

$$H(P, Q) = -\sum_{x} P(x) \log Q(x)$$

When $Q = P$ (the model predicts exactly right), cross-entropy equals entropy, its minimum possible value. When Q diverges from P, cross-entropy grows. It grows in a specific, asymmetric way: the model pays a small penalty for being uncertain, and a large penalty for being confidently wrong.

This is the loss function used to train classifiers. In binary classification, P is the true label (0 or 1) and Q is the model's predicted probability $\hat{y}$. The **binary cross-entropy loss** is:

$$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

When $\hat{y}$ is close to y, L is small. When $\hat{y}$ is far from y (say $y = 1$ but $\hat{y} = 0.01$), $\log(0.01)$ is a large negative number and L explodes. Cross-entropy punishes confident wrong predictions far more heavily than uncertain ones. That's the behaviour wanted from a classification loss: a model that says "I'm 99% sure this is fraud" and is wrong should be penalised much harder than one that shrugged and said "meh, maybe, 50/50."

Cross-entropy and entropy are linked by an identity that matters more than you might think (if you're new to this, you'd have no reason to think it matters at all):

$$H(P, Q) = H(P) + \mathrm{KL}(P \,\|\, Q)$$

Cross-entropy equals the true entropy plus a penalty term (KL divergence) for using the wrong distribution. $H(P)$, the entropy of the true labels, is fixed. It doesn't depend on the model's parameters. So minimising cross-entropy during training is mathematically identical to minimising KL divergence between the model's predictions and the truth.

5.2.3 KL Divergence: How Far Is Q From P?

KL divergence (Kullback-Leibler divergence) is a directed measure of how different a distribution Q is from a reference distribution P:

$$\mathrm{KL}(P \,\|\, Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)}$$

Three properties matter in practice. First, $\mathrm{KL}(P \,\|\, Q) \geq 0$ always, with equality only when $P = Q$. It behaves somewhat like a distance. Second, and unlike a true distance, it is *not* symmetric: $\mathrm{KL}(P \,\|\, Q) \neq \mathrm{KL}(Q \,\|\, P)$ in general. It's a directed divergence, not a metric.

Third, it is undefined (infinite) at any point where $Q(x) = 0$ but $P(x) > 0$. Q must cover the support of P, or the divergence blows up.

KL divergence shows up constantly beyond the classification loss it's hiding inside. A **variational autoencoder's** loss includes a KL term that pulls the learned latent distribution towards something close to a standard Gaussian. **RLHF** constrains a fine-tuned language model with a KL penalty against the base model. That keeps optimisation for the reward signal from drifting the model into degenerate, reward-hacking behaviour. **Knowledge distillation** trains a small student model to minimise its KL divergence from a large teacher model's output distribution. **TRPO**, and the KL-penalty variant of **PPO**, use KL divergence between successive policies to keep updates from being too large. In every one of these, the same asymmetric, always-non-negative quantity from the equation above is doing the work.

> **i** What Is a Latent Distribution?
>
> A **latent** variable is one the model never observes directly. It's the model's own internal, compressed summary of the input, learned rather than measured. A variational autoencoder compresses each input (an image, say) down to a small vector of numbers, its latent representation. It then tries to reconstruct the original input from just that vector. The **latent distribution** is the probability distribution over those compressed vectors that the model learns to produce. Pulling it towards a standard Gaussian, via the KL term mentioned above, keeps that compressed space well-behaved: smooth, centred, and usable for generating new examples by sampling from it later. Book 2 covers variational autoencoders and their architecture in full.

5.2.4 Mutual Information: How Much Does Knowing X Tell You About Y?

Mutual information measures how much knowing one variable reduces uncertainty about another:

$$I(X;Y) = H(X) - H(X \mid Y) = H(Y) - H(Y \mid X)$$

Equivalently, $I(X;Y) = \mathrm{KL}(P(X,Y) \,\|\, P(X)P(Y))$: mutual information is the KL divergence between the actual joint distribution of X and Y and the joint distribution they would have if independent. When $I(X;Y) = 0$, X and Y are independent: knowing one tells you nothing about the other. When $I(X;Y)$ is large, they're strongly dependent.

The practical use is feature selection:
`sklearn.feature_selection.mutual_info_classif` ranks candidate features by how much each reduces uncertainty about the label. Chapter 3 built correlation and covariance out of sums of products of deviations from the mean. That machinery only ever sees

straight-line relationships. Mutual information asks the same practical question, does this feature relate to the label, but answers it from entropy instead of deviations. That different foundation is exactly why it succeeds where correlation fails. Nothing in its construction assumes the relationship is linear, so it catches dependence covariance is structurally blind to. Mutual information also underlies the **information bottleneck** framework, for understanding what a neural network's internal layers retain. It is also behind contrastive methods like SimCLR and CLIP. Their InfoNCE loss maximises a lower bound on the mutual information between paired inputs: two augmented views of one image (SimCLR), or an image and its caption (CLIP).

5.2.5 How the Four Fit Together

$H(P)$	how uncertain is the truth?
$H(P, Q)$	how well does the model Q fit P?
$\mathrm{KL}(P \,\|\, Q) = H(P, Q) - H(P)$	how far is Q from P?
$I(X; Y) = H(Y) - H(Y \mid X)$	what does X tell you about Y?

Entropy is the baseline difficulty of a problem. Cross-entropy is what a model is being scored on. KL divergence is the gap between the two. Mutual information asks the same underlying question (how much does one thing tell you about another) of two *variables* rather than of a model against the truth.

5.3 Trade-offs and Limits

Entropy and Gini impurity are the two candidates for measuring node impurity in decision trees. In practice they produce nearly identical splits. Entropy is more theoretically grounded: it's the same quantity used everywhere else in this chapter. Gini is a little cheaper to compute, since it needs no logarithm, and scikit-learn defaults to it. The choice rarely changes which feature a tree picks.

Cross-entropy versus mean squared error for classification is not such a close call. Cross-entropy is the mathematically correct loss: it falls directly out of the likelihood of a Bernoulli or categorical distribution. MSE applied to classification produces saturated gradients near 0 and 1, exactly where a classifier most needs strong gradient signal to fix a confident, wrong prediction. Cross-entropy should always be used for classification. Reaching for MSE there is a common and avoidable mistake.

KL divergence's asymmetry could be seen as a flaw. Better to treat it as a feature to be deliberate about. `KL(P||Q)` and `KL(Q||P)` answer different questions. Mixing them up produces subtly wrong regularisation or drift-detection behaviour. Always be explicit about

which distribution is "truth" (the first argument) and which is the "approximation" (the second). Mutual information's main limitation is that, unlike a correlation coefficient's fixed $[-1, 1]$ range, it has no fixed scale: its ceiling is the label's entropy, which changes from dataset to dataset, and there's no universal "this counts as strong" threshold. It is safe to use for *ranking* features against each other within one dataset and one run, but not as an absolute quality score comparable across different problems.

5.4 Procedures

Procedure 5.1 (Compute Entropy of a Distribution). *When to use:* summarising how uncertain or spread out a distribution is: class balance in a dataset, or a model's predicted class probabilities.

Listing 5.1 Entropy from scratch, and the scipy equivalent

```
import numpy as np
import scipy.stats as stats

def entropy(p, eps=1e-12):
    p = np.clip(np.asarray(p, dtype=float), eps, 1.0)
    return -np.sum(p * np.log2(p))  # bits

# Imbalanced binary label
class_balance = np.array([0.9, 0.1])
# Low -- easy to predict the majority class
print(entropy(class_balance))

# scipy equivalent (natural log by default -- pass base=2
# to match bits)
print(stats.entropy(class_balance, base=2))
```

Common mistake: passing raw counts instead of a normalised distribution. `entropy([90, 10])` is wrong. Normalise first with `p / p.sum()`, or let `scipy.stats.entropy` do it (it normalises automatically, unlike a hand-rolled version).

Procedure 5.2 (Compute Cross-Entropy Loss). *When to use:* training or evaluating a classifier, or sanity-checking predicted probabilities against ground truth.

Listing 5.2 Binary cross-entropy from scratch and via sklearn

```
import numpy as np
from sklearn.metrics import log_loss

def binary_cross_entropy(y_true, y_pred, eps=1e-12):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -np.mean(
        y_true * np.log(y_pred)
        + (1 - y_true) * np.log(1 - y_pred)
    )

y_true = np.array([1, 0, 1, 1])
y_pred = np.array([0.9, 0.2, 0.6, 0.95])
print(binary_cross_entropy(y_true, y_pred))

# sklearn handles the multi-class case and the clipping for you
print(log_loss(y_true, y_pred))
```

Here is a rule of thumb for reading raw loss values (natural log, nats). Around 0.69 is no better than a coin flip on a balanced binary problem ($-\log(0.5)$); below 0.1 is confident and mostly correct; above 2.0 means the model is confidently wrong on a meaningful fraction of examples.

Common mistake: feeding raw logits into a loss function that expects probabilities, or vice versa. `sklearn.metrics.log_loss` expects probabilities in $[0, 1]$. PyTorch's `nn.CrossEntropyLoss` expects raw logits and applies softmax internally. Passing already-softmaxed probabilities double-applies the transform and silently produces a plausible-looking but wrong loss.

Procedure 5.3 (Compute KL Divergence Between Two Distributions). *When to use:* comparing a predicted distribution against a reference or prior, checking a feature's distribution for drift over time, or implementing a VAE or RLHF penalty term.

Listing 5.3 KL divergence from scratch, showing its asymmetry

```
import numpy as np
import scipy.stats as stats

def kl_divergence(p, q, eps=1e-12):
    p = np.clip(np.asarray(p, dtype=float), eps, 1.0)
    q = np.clip(np.asarray(q, dtype=float), eps, 1.0)
    return np.sum(p * np.log2(p / q))
```

```
p = np.array([0.5, 0.3, 0.2])
q = np.array([0.3, 0.3, 0.4])
print(kl_divergence(p, q))          # KL(P || Q)
# KL(Q || P) -- different! not symmetric
print(kl_divergence(q, p))

# scipy: entropy(p, q) computes KL(p || q) when q is provided
print(stats.entropy(p, q, base=2))
```

Common mistake: treating KL divergence as an interchangeable distance metric and swapping the arguments freely. `KL(P||Q) != KL(Q||P)`, so always be explicit about which is truth (first argument) and which is approximation (second). Separately, KL divergence is infinite wherever `q` is zero and `p` isn't, which is the practical reason for the `eps` clipping above.

Procedure 5.4 (Rank Features by Mutual Information). *When to use:* selecting predictive features when a non-linear relationship to the label is suspected, one that a correlation matrix would miss entirely.

Listing 5.4 Ranking features by mutual information rather than correlation

```
from sklearn.feature_selection import (
    mutual_info_classif,
    mutual_info_regression,
)
import pandas as pd

# Classification target
mi_scores = mutual_info_classif(X, y, random_state=42)
ranking = (
    pd.Series(mi_scores, index=X.columns)
    .sort_values(ascending=False)
)
print(ranking)

# Regression target
mi_scores_reg = mutual_info_regression(
    X, y_continuous, random_state=42
)

# Keep only the top-K informative features
top_k_features = ranking.head(10).index
X_selected = X[top_k_features]
```

Common mistake: comparing mutual information scores across different runs or datasets as if they sat on a fixed, interpretable scale like a correlation's $[-1, 1]$ range. MI is non-negative, but it has no fixed scale. For a class label, its ceiling is the label's entropy, which changes from dataset to dataset. Use it only to rank features relative to each other within the same run, not as an absolute quality threshold.

Procedure 5.5 (Diagnose a Cross-Entropy Loss That Won't Go Down). *When to use:* training a classifier whose loss is stuck, exploding, or has gone to `nan`.

Symptom	Most likely cause	Fix
Loss is `nan`	`log(0)` from an unclipped prediction of exactly 0 or 1	Clip predictions to $[\epsilon, 1-\epsilon]$, or use a numerically stable framework loss instead of a manual log
Loss stuck near $-\log(1/K)$	Model is predicting close to uniform, hasn't learned anything yet	Check the learning rate. Check labels aren't shuffled or misaligned
Loss decreases, then suddenly spikes	An extreme confident-and-wrong prediction on a mislabelled or outlier example	Inspect the highest per-example losses. Check for label noise
Train loss low, validation loss high	Overfitting, not an information-theory problem	Regularisation, covered later in this book

5.5 Worked Example

The companion notebook implements all four quantities from scratch in numpy and checks that they behave the way the theory in this chapter predicts, rather than deriving anything new.

It starts with entropy over a four-sided die in three configurations, using Listing 5.1's function:

Listing 5.5 Entropy of a fair die, a biased die, and a point mass

```
fair_die     = np.array([0.25, 0.25, 0.25, 0.25])
biased_die   = np.array([0.70, 0.10, 0.10, 0.10])
point_mass   = np.array([1.00, 0.00, 0.00, 0.00])

distributions = [
    ('Fair die', fair_die),
    ('Biased die', biased_die),
    ('Point mass', point_mass),
]
for name, dist in distributions:
    print(f"{name:12s}: H = {entropy(dist):.4f} bits")
```

The computed results: the fair die has $H = 2.0000$ bits, exactly $\log_2(4)$, the maximum possible for four outcomes. The biased die has $H = 1.3568$ bits, less uncertainty, since one outcome dominates. The point mass has $H = 0.0000$ bits, nothing left to be uncertain about. All three land exactly where the Exposition section's theory says they should.

Next, the notebook simulates a single true label $y = 1$ and computes binary cross-entropy at three predicted probabilities:

Listing 5.6 Cross-entropy loss at three prediction confidences, same true label

```
y_true_single = np.array([1.0])
l_correct = binary_cross_entropy(y_true_single, np.array([0.99]))
l_unsure = binary_cross_entropy(y_true_single, np.array([0.50]))
l_wrong = binary_cross_entropy(y_true_single, np.array([0.01]))

print(f"Confident and correct   (ŷ=0.99): loss = {l_correct:.4f}")
print(f"Uncertain               (ŷ=0.50): loss = {l_unsure:.4f}")
print(f"Confident and WRONG     (ŷ=0.01): loss = {l_wrong:.4f}")
```

The computed losses are 0.0101 nats for the confident-correct prediction, 0.6931 nats for the maximally uncertain one, and 4.6052 nats for the confident-and-wrong one. The confidently wrong prediction costs **458 times** as much as the confident-correct one, and about **6.6 times** as much as the maximally uncertain one. That's a sharply asymmetric penalty. It's exactly the behaviour that makes cross-entropy, not MSE, the correct loss for classification.

The notebook then verifies the identity $H(P, Q) = H(P) + \mathrm{KL}(P \| Q)$ directly, using $P = [0.50, 0.30, 0.15, 0.05]$ and $Q = [0.30, 0.30, 0.20, 0.20]$:

Listing 5.7 Verifying the cross-entropy/entropy/KL-divergence identity numerically

```
P = np.array([0.50, 0.30, 0.15, 0.05])
Q = np.array([0.30, 0.30, 0.20, 0.20])

kl_pq = kl_divergence(P, Q)
kl_qp = kl_divergence(Q, P)
h_p = entropy(P)
h_pq = cross_entropy_dist(P, Q)
```

The computed values: $\mathrm{KL}(P\|Q) = 0.2062$ bits and $\mathrm{KL}(Q\|P) = 0.2619$ bits. That confirms the asymmetry claimed in the Exposition. $H(P) = 1.6477$ bits. $H(P) + \mathrm{KL}(P\|Q) = 1.8540$ bits. That matches the directly computed $H(P, Q) = 1.8540$ bits exactly. The identity linking cross-entropy, entropy, and KL divergence isn't an approximation. It holds bit for bit.

Finally, the notebook builds a small synthetic dataset designed to expose correlation's blind spot: `feature_linear`, whose *mean* depends on the label; `feature_nonlinear`, whose *variance* (not mean) depends on the label; and `feature_noise`, unrelated to either. Ranking all three by both |correlation| and mutual information, using Listing 15.3, produces:

Feature	|correlation|	Mutual information (nats)
`feature_nonlinear`	0.0161	0.3377
`feature_linear`	0.7047	0.3303
`feature_noise`	0.0075	0.0430

Correlation ranks `feature_nonlinear` (0.0161) as statistically indistinguishable from pure noise (0.0075). Its mean doesn't move with the label, only its spread does, and Pearson correlation is blind to that. Mutual information ranks it as the single *most* informative feature in the dataset, fractionally ahead of `feature_linear`. It measures the full distributional dependence between feature and label, not just a linear one. This is the practical payoff of everything in this chapter's Exposition section: mutual information catches exactly the relationship correlation is mathematically incapable of seeing.

This chapter has no AWS SageMaker Notes sidebar. Every computation above runs on CPU in milliseconds. For mutual-information feature selection at a scale that no longer fits comfortably in a notebook, ***`mutual_info_classif`*** *can run inside a SageMaker Processing job. That pattern reappears in this book once a chapter has a dataset large enough to need it.*

5.6 History and Further Reading

Murphy's Introduction, Ch. 6, gives the full treatment of entropy, cross-entropy, KL divergence, mutual information, and their connection to data compression. It's the natural next stop after this chapter (Murphy 2022). Burkov covers cross-entropy specifically as a loss function, concisely and directly tied to the classification setting this chapter used to motivate it (Burkov 2019, 34–40). Shannon's original 1948 paper is the source of essentially everything in this chapter. It's unusually readable for a foundational paper of this significance (Shannon 1948). Christopher Olah's essay "Visual Information Theory" (colah.github.io, 2015) is, informally, the best available visual intuition for entropy, cross-entropy, and KL divergence. It's worth reading alongside this chapter's equations, rather than instead of them.

5.7 How This Connects

This chapter builds on Chapter 3's treatment of probability and distributions. Information content is only meaningful once "probability of an event" is already a settled idea. It enables the loss functions behind decision trees and neural network training (Chapter 11 and later chapters on classical and deep models). It also underpins variational autoencoders and RLHF (covered in Book 2) and the feature-selection techniques that recur throughout the data preparation chapters of this book. It is most often confused with a mix-up between entropy and cross-entropy. Entropy measures the uncertainty of the *true* distribution alone. Cross-entropy measures the cost of using a *predicted* distribution in its place. They differ by exactly the KL divergence between the two. Training a classifier means minimising cross-entropy. That indirectly minimises the KL term, since entropy is fixed.

5.8 Exercises

1. **[W]** Without computing anything, rank a fair four-sided die, a die with probabilities $[0.70, 0.10, 0.10, 0.10]$, and a die that always lands on the same face, from highest to lowest entropy, and justify the ranking in one sentence.
2. **[W]** Explain in your own words why $\mathrm{KL}(P\|Q) \neq \mathrm{KL}(Q\|P)$ in general. Give a one-sentence rule for keeping "truth" and "approximation" straight when calling a KL divergence function.
3. **[A]** Using Listing 5.2, compute the binary cross-entropy loss for `y_true = [1, 1, 0, 0]` against two prediction sets: `y_pred_a = [0.6, 0.6, 0.4, 0.4]` and `y_pred_b = [0.99, 0.51, 0.49, 0.01]`. Both models get every prediction "on the correct side of 0.5." Explain why their losses differ anyway.

4. **[A]** Using Listing 5.5's pattern, compute the entropy of a five-outcome distribution $[0.4, 0.4, 0.1, 0.05, 0.05]$ in bits. State how far it falls below the five-outcome maximum of $\log_2(5)$.
5. **[A]** Reproduce Listing 5.7's identity check using $P = [0.6, 0.4]$ and $Q = [0.5, 0.5]$, and report $\mathrm{KL}(P\|Q)$, $\mathrm{KL}(Q\|P)$, and whether $H(P, Q) = H(P) + \mathrm{KL}(P\|Q)$ still holds.
6. **[E]** A colleague proposes ranking every feature in a 200-column dataset by mutual information against the label and keeping only those above a score of 0.1, arguing "0.1 is a reasonable bar, like a correlation of 0.1." Using this chapter's Trade-offs section, explain what's wrong with treating 0.1 as a fixed, portable threshold.
7. **[E]** Design a synthetic feature, distinct from `feature_nonlinear` in the Worked Example, whose mean is unrelated to a binary label but that is still informative about it (i.e., correlation should be near zero but mutual information should be meaningfully above the noise floor). Describe the mechanism in words. You don't need to run it.
8. **[R]** Read Shannon's original 1948 paper, or at minimum its introduction. Identify one idea in it that this chapter did not cover, and explain in a few sentences how it relates to (or extends) the entropy/cross-entropy/KL/mutual-information picture built here.

5.9 Mental Model

Entropy is the baseline: how hard is this problem, before any model touches it? Cross-entropy is the scorecard: how well is a model doing against that baseline? KL divergence is the gap between the two: how much worse than optimal is the model being? Mutual information asks the same question of two variables instead of a model and the truth: how much does one tell you about the other? When a classifier's cross-entropy loss converges to $H(P)$, the model has learned everything there is to learn from the data. No amount of further training can close a gap that no longer exists.

6 Python Data Science Ecosystem

Python was not designed for data science. It won anyway. Not because of the language itself but because of the layered stack of libraries built on top of it. Each layer solves a specific problem the one below it doesn't. NumPy makes numerical computation fast. pandas adds labels and structure to tabular data. matplotlib and seaborn turn tables into pictures. scipy supplies the statistical machinery. scikit-learn wraps every classical algorithm behind one consistent interface. None of these libraries are magic. Together they form the working vocabulary of everything that follows in this book. This chapter is a map of that stack: what each layer does, why it exists, and how the pieces compose.

6.1 Motivation

Every later chapter in this book assumes fluency with this stack: not memorised syntax, but an understanding of which library to reach for and why. A Solutions Architect who understands the layering can diagnose problems quickly. A slow `for` loop over a NumPy array is immediately suspect. A sklearn `DataConversionWarning` is immediately traceable to array shape. A matplotlib customisation that seaborn's API doesn't expose is immediately solvable by dropping one level down. Without that mental model, the same problems turn into hours of aimless searching. The goal here isn't full API coverage. Appendix A has that. It's the *why* behind the design of each layer, so the *what* becomes easy to look up when needed.

6.2 Exposition

6.2.1 The Layered Stack

The Python data science stack is layered. Each layer builds on the one below it. scikit-learn, XGBoost, and statsmodels sit at the top, providing ML algorithms. pandas sits below that, providing labelled tabular data. matplotlib, seaborn, and plotly provide visualisation. scipy provides general scientific computing. NumPy provides fast array operations. CPython, with its C extensions, is the engine underneath all of it.

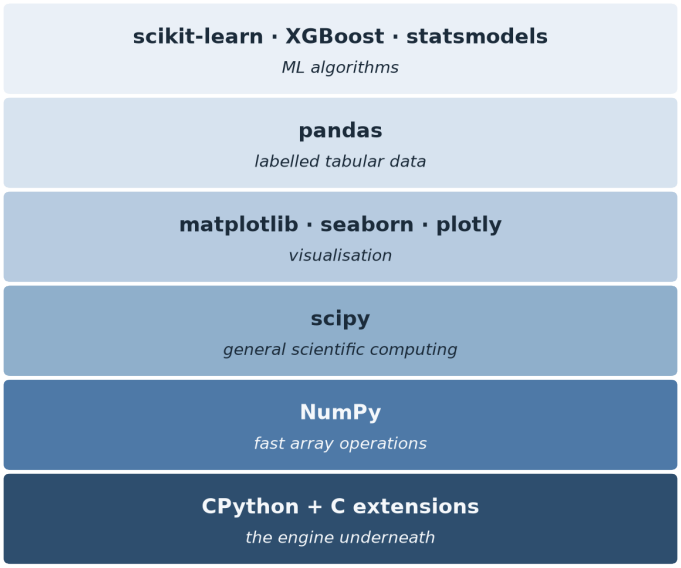


Figure 6.1: The Python data science stack: each layer builds on the one below it.

You don't need to understand C to use NumPy. But you do need to understand NumPy to use pandas effectively. And you need pandas to use sklearn well. The abstractions are leaky in useful ways. pandas is built directly on NumPy arrays. Knowing that explains a great deal of pandas' behaviour: why operations are fast, why `.values` and `.to_numpy()` exist, and why sklearn ultimately wants arrays, not DataFrames.

6.2.2 NumPy: Fast Arrays

Pure Python loops are slow for numerical work. Every iteration of `for item in my_list: result += item * 2` forces the interpreter to check the type of `item`, look up the `*` operator, allocate a new integer object, and manage garbage collection. That overhead dominates once a list reaches a million elements. NumPy solves this by storing data in contiguous blocks of memory with a single fixed type: a `float64` array is 64 bits per element, laid out in order. It implements operations on that memory in compiled C. That typically runs somewhere between 50 and 200 times faster than the equivalent Python loop, for simple elementwise operations like this.

The feature that makes this usable rather than just fast is **broadcasting**: NumPy applies operations between arrays of matching or compatible shapes without an explicit loop. `array * 2`, `array_a + array_b`, `matrix @ vector` all work element-wise or via broadcasting rules. All of them run in compiled code.

> **What Is Broadcasting?**
>
> Broadcasting is NumPy's rule for combining arrays of different shapes without writing a loop to line them up by hand. The simplest case: `array + 5` adds 5 to every element. NumPy treats the single number 5 as if it were stretched into an array the same shape as `array`, without copying it in memory. The same idea extends to two arrays: adding a `(3,)` array to a `(4, 3)` array adds that one row to every one of the four rows. NumPy lines up their shapes from the right, and stretches any dimension that's size 1, or missing, to match. Shapes that can't be lined up this way raise a `ValueError`. Shapes that line up by accident broadcast without complaint, so a wrong-shaped array can still give a wrong answer silently.

Boolean indexing follows the same principle: `array[array > threshold]` filters without a loop, by first constructing a boolean mask (`array > threshold`) and then using it to select.

6.2.3 pandas: Labelled Tabular Data

NumPy arrays have no column names, no row labels, and no native concept of mixed types across columns. Every real dataset has all three. pandas supplies two structures on top of NumPy: a **Series**, a one-dimensional labelled array representing a single column, and a **DataFrame**, a two-dimensional labelled table whose columns can each hold a different type. The central concept is the **index**: every row carries a label, not just a position. That lets pandas align data from different sources automatically, support time-series-specific operations, and avoid the off-by-one errors that plague pure positional indexing. Because most DataFrame columns are NumPy arrays under the hood, pandas operations inherit NumPy's speed. Any pandas Series can be dropped back down to a raw array with `.to_numpy()` whenever raw-array behaviour is needed.

6.2.4 matplotlib: The Low-Level Plotting Layer

matplotlib is the foundation of Python visualisation, and is deliberately imperative. It gives precise control over every element of a plot: axes, ticks, labels, colours, line styles, text. But it requires each of them to be spelled out, step by step. Every other visualisation library in the stack is built on top of it. When seaborn returns a plot, it is returning a matplotlib figure. When pandas' `.plot()` is called, it is calling matplotlib internally. Fluency with matplotlib means any chart, regardless of which higher-level library generated it, can always be customised further.

6.2.5 seaborn: Statistical Visualisation

seaborn is a high-level wrapper around matplotlib, purpose-built for statistical data visualisation. It understands DataFrames natively, ships with better-looking defaults, and has built-in support for the chart types data scientists reach for constantly: distribution plots, regression plots, categorical comparisons, heatmaps, pairplots. In practice, matplotlib is for *drawing*, seaborn is for *exploring data*. For exploratory data analysis, seaborn is almost always the right starting point. For fine-grained custom charts, dropping to matplotlib is the right move.

6.2.6 scipy: Scientific Computing

scipy sits alongside NumPy and supplies algorithms that pure array operations can't handle efficiently on their own: statistical tests, optimisation, signal processing, additional linear algebra, interpolation, integration. In day-to-day data science work, the module used most is `scipy.stats` (t-tests, chi-squared tests, and probability distributions), with `scipy.optimize` making occasional appearances (Chapter 4 covers optimisation in depth).

6.2.7 scikit-learn: The ML Algorithm Layer

scikit-learn is probably the most considered library in the ecosystem. Its defining feature is a **consistent API** that every estimator follows: `.fit(X, y)` learns from data. `.predict(X)` generates predictions for supervised models. `.transform(X)` transforms data for preprocessing steps. `.fit_transform(X)` does both in one call. `.score(X, y)` evaluates performance. Once `LogisticRegression` is understood, `RandomForestClassifier` is essentially already understood: same `.fit()`, same `.predict()`, same `.score()`. This consistency pays off more than it looks. The cost of learning a new algorithm in sklearn is mostly the cost of understanding the algorithm itself. sklearn also supplies preprocessing utilities (scaling, encoding), cross-validation, evaluation metrics, and pipelines that chain multiple steps into a single object.

6.2.8 Jupyter Notebooks: Literate Computing

A Jupyter notebook mixes live code, output (including charts), and markdown text in one document. The idea is older than Jupyter itself. Donald Knuth called this literate programming in 1984: write the code and the explanation of the code as a single interleaved document, meant to be read by a person first and a computer second. A notebook is that idea, running, decades later. It is the standard environment for data science because it makes exploration natural: run a cell, see the result, adjust, run again. A notebook doubles as a communication artefact. It can be shared with someone who doesn't read Python, and they

can still follow the narrative and understand the findings. The trade-off is that notebooks encourage non-linear execution: running cells out of order. That produces hidden state bugs that aren't visible in the code as written. The discipline that prevents this: restart the kernel and run all cells top to bottom before treating any result as final.

6.3 Trade-offs and Limits

This stack excels at rapid prototyping and exploration, and at tabular data of any size that fits in memory. It's good for sharing work with colleagues through notebooks. It also gives access to almost every classical ML algorithm ever published through one consistent interface. It struggles in three situations. Datasets larger than available RAM, where Dask, Spark, or a cloud-native tool is needed instead. Real-time production serving, where a dedicated serving framework replaces ad hoc notebook code (Book 2, Chapter 23 covers this in depth). Mobile or edge deployment, where LiteRT (formerly TensorFlow Lite) or ONNX are the right tools rather than the full stack. None of these are flaws in the libraries. They're outside the problem this stack was built to solve: interactive, in-memory, exploratory analysis and modelling.

6.4 Procedures

Procedure 6.1 (Install and Verify the Stack). *When to use:* at the start of any new project or environment, before writing analysis code.

Listing 6.1 Installing the core stack and verifying the active environment has it

```
# Option A -- pip (minimal, exact control)
# pip install numpy pandas matplotlib seaborn scipy \
#     scikit-learn jupyter

# Option B -- conda (recommended for reproducibility)
# conda create -n dsenv python=3.11
# conda activate dsenv
# conda install numpy pandas matplotlib seaborn scipy \
#     scikit-learn jupyter

# Option C -- check an existing environment
import numpy, pandas, matplotlib, seaborn, scipy, sklearn
print(numpy.__version__, pandas.__version__, sklearn.__version__)
```

What to look for: a `ModuleNotFoundError` almost always means the wrong environment is active, not that the package is missing system-wide. Check `which python` / `conda info --envs` before reinstalling anything.

Procedure 6.2 (Convert Between NumPy Arrays and pandas DataFrames). *When to use:* scikit-learn's estimators accept NumPy arrays; real data usually lives in a DataFrame first, so conversion happens at the fit/predict boundary.

Listing 6.2 Converting between DataFrame, Series, and NumPy array representations

```
import numpy as np
import pandas as pd

# DataFrame -> NumPy array
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
arr = df.to_numpy()            # (3, 2)

# NumPy array -> DataFrame
arr = np.array([[1, 2], [3, 4], [5, 6]])
df = pd.DataFrame(arr, columns=['col_a', 'col_b'])

# Series <-> 1D array
arr_1d = df['col_a'].to_numpy()
s = pd.Series(arr_1d, name='col_a')
```

Common mistake: scikit-learn raises a warning if a model is fit on a DataFrame with named features and later passed a plain array, or vice versa. Pick one representation per pipeline and stay consistent through the whole `fit`/`predict` cycle.

Procedure 6.3 (Plot Directly from a DataFrame). *When to use:* quick exploratory plots, before setting up matplotlib axes by hand is worth the effort.

Listing 6.3 Plotting straight off a DataFrame for fast exploratory charts

```
import matplotlib.pyplot as plt

# line
df['value'].plot(title='My Value Over Time')
# bar
df.groupby('category')['value'].mean().plot(kind='bar', rot=0)
# histogram
df['age'].plot(kind='hist', bins=20, edgecolor='white')
```

```
# scatter
df.plot(kind='scatter', x='feature_a', y='feature_b', alpha=0.5)

plt.tight_layout()
plt.show()
```

Procedure 6.4 (Use scikit-learn's Consistent fit/predict/score API). *When to use:* every time any sklearn estimator is fit. The five-method pattern never changes across algorithms.

Listing 6.4 The fit / predict / score pattern every sklearn estimator follows

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)              # learn
y_pred = model.predict(X_test)           # predict
print(accuracy_score(y_test, y_pred))    # evaluate
```

Common mistake: fitting a scaler or encoder on `X_test` rather than only on `X_train`. The scaler must learn its mean and standard deviation exclusively from training data, then apply that same transform to test data:

Listing 6.5 Fitting a scaler once on train data, applying it to both splits

```
scaler = StandardScaler()
# fit ON train, transform train
X_train_scaled = scaler.fit_transform(X_train)
# transform test with the SAME scaler
X_test_scaled = scaler.transform(X_test)
```

Fitting the scaler on `X_test` (or on the full dataset before splitting) leaks test-set statistics into training and silently inflates reported performance. The model looks better than it is, with no error raised to signal it.

Procedure 6.5 (Diagnose Common Stack Errors). *When to use:* when an error message doesn't obviously point at its own cause.

Error / symptom	Most likely cause	Fix
`ModuleNotFoundError: No module named 'numpy'`	Wrong environment active	Activate the right env, then reinstall
`ValueError: Input contains NaN`	sklearn received missing values	Clean data first (Chapter 7)
`DataConversionWarning: A column-vector y was passed`	`y` has shape `(n, 1)` not `(n,)`	`y.ravel()` or `y.squeeze()`
Plot is blank / not showing	`plt.show()` called before the data-plotting calls	Move plotting calls above `plt.show()`
seaborn `hue` not working	The hue column contains `NaN` values	`df.dropna(subset=['hue_col'])` first

6.5 Worked Example

The tutorial notebook behind this chapter is a tour through the whole stack, on two real, bundled datasets: seaborn's Titanic passenger list and scikit-learn's iris flower measurements. Three moments from it are worth walking through in detail, because each one makes a claim from the Exposition concrete with real numbers.

The first is the NumPy-versus-Python-loop speed claim. Doubling one million elements with a Python list comprehension versus NumPy's vectorised multiplication, timed directly:

Listing 6.6 Measuring the vectorisation speedup directly rather than asserting it

```
import numpy as np
import timeit

N = 1_000_000
py_list = list(range(N))
np_array = np.arange(N, dtype=np.float64)

t_py = timeit.timeit(
    '[x * 2 for x in py_list]', globals=globals(), number=5
) / 5
t_np = timeit.timeit(
```

```
    'np_array * 2', globals=globals(), number=5
) / 5
print(f'Python loop: {t_py*1000:.2f} ms')
print(f'NumPy:       {t_np*1000:.3f} ms')
print(f'Speedup:     {t_py/t_np:.0f}x')
```

On this machine, the Python loop takes roughly **27.7 ms** and the NumPy operation roughly **0.24 ms**, a **117x** speedup, well within the "50-200x" range this chapter's Exposition claims, and a difference that only grows as array size grows further.

The second moment combines pandas and scipy on the Titanic dataset (891 passengers, 15 columns). A grouped aggregation answers a real question (did survival rate depend on passenger class and sex?) in one line:

Listing 6.7 A pivot table answering a two-dimensional question in one call

```
pivot = df.pivot_table(
    values='survived',
    index='pclass',
    columns='sex',
    aggfunc='mean',
)
print(pivot.round(2))
```

The result: first-class women survived at **97%**, first-class men at **37%**. By third class those numbers fall to **50%** and **14%** respectively. Class and sex both mattered, and mattered together. A single-dimension `groupby` would have hidden that. A natural follow-up question (did survivors pay more?) is answered with a two-sample t-test:

Listing 6.8 A one-line hypothesis test on real fare data

```
from scipy import stats

survivors = df[df['survived'] == 1]['fare'].dropna()
non_survivors = df[df['survived'] == 0]['fare'].dropna()

t_stat, p_value = stats.ttest_ind(survivors, non_survivors)
print(f'Survivor mean fare:     £{survivors.mean():.2f}')
print(f'Non-survivor mean fare: £{non_survivors.mean():.2f}')
print(f't = {t_stat:.2f}, p = {p_value:.2e}')
```

Survivors paid **£48.40** on average against **£22.12** for non-survivors, with `t = 7.94` and `p ~ 6.1e-15`, from the default pooled-variance test. (Chapter 9's Welch test, which doesn't assume equal variances, gives `p ~ 2.7e-11`.) That's far below the usual 0.05 threshold, so the fare gap is not plausibly due to chance. Fare correlates strongly with class, and class correlates strongly with survival. So this single number is surfacing the same class effect the pivot table showed directly. That's a reminder that a significant p-value tells you a difference is unlikely to be chance, not what's causing it.

The third moment is the scikit-learn API pattern, applied to the classic iris dataset (150 flowers, 4 features, 3 species):

Listing 6.9 fit / predict / score on a new dataset, same pattern as Listing 6.4

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = load_iris(as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.2, random_state=42
)

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print(f'Accuracy: {accuracy_score(y_test, y_pred):.0%}')
```

With 120 training and 30 test samples, this particular split scores **100% accuracy**. Iris is a famously easy, well-separated dataset. That's exactly why it's a good first dataset for learning the API pattern, rather than learning about model performance. Chapter 11 returns to k-Nearest Neighbours, on harder data, where accuracy stops being a foregone conclusion.

6.6 History and Further Reading

Thakur's Ch. 1 sets up the working environment used throughout his book. It's a good companion to this chapter's Procedure 6.1 (Thakur 2020). Burkov's Ch. 2 gives the same core concepts, vectors, matrices, functions, from the mathematical-notation side rather than the code side. It's useful for connecting this chapter to Chapter 3's formalism (Burkov

2019). Beyond the bibliography, the official NumPy quickstart at numpy.org and pandas' "10 Minutes to pandas" guide are both worth reading directly. They're as short as they claim to be, and more precise on edge cases (broadcasting rules, index alignment) than any secondary summary.

6.7 How This Connects

This chapter builds on ordinary Python fluency (lists, dicts, functions, `pip`), which this book treats as a prerequisite rather than a chapter of its own. It enables essentially everything that follows. Every remaining chapter in this book uses NumPy, pandas, and scikit-learn as its working vocabulary, the way Chapter 2 used `pandas` and `sklearn.model_selection` without pausing to introduce them. It is often confused with PyTorch and TensorFlow. Those are deep learning frameworks, covered starting in Book 2. They sit *alongside* this stack rather than inside it: a production system often uses pandas for data wrangling and scikit-learn for a baseline model, even when the final model is a neural network.

6.8 Exercises

1. **[W]** Using the layered-stack idea from this chapter's Exposition, explain in one or two sentences why learning pandas well requires understanding NumPy, but not the other way around.
2. **[W]** State in one sentence each what `.fit()`, `.transform()`, and `.predict()` do in the scikit-learn API pattern. Identify which of the three should never be called on test data.
3. **[A]** Using Listing 6.6 as a template, measure the speedup of `np.sqrt` over an equivalent Python loop (using `math.sqrt`) on an array of 500,000 elements, and report the ratio you observe.
4. **[A]** Using Listing 6.7's approach, compute survival rate by passenger class **and** `embark_town` instead of class and sex. Report which single (class, port) combination has the lowest survival rate.
5. **[A]** Reproduce Listing 6.8, but compare passenger **age** between survivors and non-survivors instead of fare. Is the difference statistically significant at $p < 0.05$? Which group is older on average?
6. **[E]** Procedure 6.4's common mistake describes fitting a scaler on test data. Write two or three sentences explaining what would go wrong with a reported accuracy number if this mistake were made silently on a real project, and why no exception would be raised to catch it.

7. [**E**] A colleague proposes dropping pandas entirely and working with raw NumPy arrays "for speed" on a project with twenty columns of mixed numeric and categorical data. Using this chapter's Trade-offs section, write the two or three sentence argument against this.
8. [**R**] Read the NumPy quickstart mentioned in this chapter's Further Reading. Find one broadcasting rule this chapter's Exposition did not discuss. Describe a data science situation where getting that rule wrong would silently produce an incorrect result rather than an error.

6.9 Mental Model

Think of the ecosystem as a toolbox with layers of abstraction. NumPy is the raw material. pandas is the workbench. matplotlib and seaborn are the drafting table. scikit-learn is the instrument kit. Going below NumPy is rarely necessary. pandas and scikit-learn will be used in essentially every project in this book. Knowing which layer a problem lives in is most of the battle in solving it quickly.

7 Data Cleaning & Wrangling

Real-world data does not arrive clean. It comes from databases with NULLs. Scrapers with encoding errors. Manual entry with typos. Merges with duplicate records. Legacy systems whose type conventions made sense years ago but not today. A model trained on top of that doesn't average the noise away. It often learns the noise *better* than the signal. Noise tends to be more systematic than it looks. Every data quality problem reduces to one of a small number of categories, no matter how varied the symptom: missing values, outliers, duplicates, wrong types, encoding errors, inconsistent categories, and schema violations. This chapter builds the diagnostic habit and the treatment toolkit for all seven. It argues that cleaning is not a chore that comes before the real work of data science. For most practitioners, on most projects, it *is* the real work.

7.1 Motivation

Surveys of data scientists put data preparation and cleaning at anywhere from about 40% to 80% of their working time. Not because they're inefficient. The work is that large relative to modelling. The reason cleaning matters as much as it does is uneven risk. A model can survive a slightly suboptimal hyperparameter. It cannot survive training on systematically corrupted labels or features. It will faithfully learn the corruption. The discipline this chapter teaches is a repeatable loop that works on any new dataset, regardless of domain, shown in Figure 7.1.

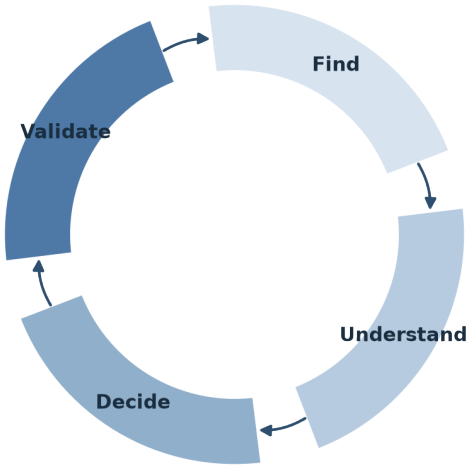


Figure 7.1: The data-cleaning loop: find, understand, decide, validate, then back to find.

That loop is not unique to data cleaning. It is a special case of an older idea: Boyd's OODA loop, observe, orient, decide, act. Find is observation: what is in the data. Understand is orientation: why it is there, what produced it. Decide keeps its name. Validate plays the role of act. It checks that the decision, once taken, worked, before the next problem gets the same treatment. Skipping straight to treatment without understanding *why* a value is missing or unusual is the most common way cleaning goes wrong. It is exactly the failure OODA warns against: acting on an assumption instead of on what the data actually shows.

A Little History: The OODA Loop

John Boyd developed the OODA loop as a US Air Force fighter pilot and military strategist, drawing on his analysis of why American F-86 pilots consistently beat the technically superior MiG-15 in Korean War dogfights. His conclusion: the pilot who cycles through observing, orienting, deciding, and acting fastest wins, regardless of whose aircraft is better. Boyd never published the idea as a single paper. It spread through a long unpublished briefing, "Patterns of Conflict," that he gave throughout the 1970s and 80s. Business strategy, cybersecurity incident response, and sports coaching have since borrowed the loop far outside its military origin, the same way this chapter borrows it for data cleaning.

7.2 Exposition

7.2.1 Missing Values

Missing data is one of three different problems, depending on the *mechanism* that produced the gap. The mechanism determines which treatments are safe:

- **MCAR (Missing Completely At Random):** the probability of a value being missing has nothing to do with any other variable: a sensor randomly dropped a reading. This is the safest case: dropping or imputing introduces no systematic bias.
- **MAR (Missing At Random):** missingness depends on *observed* variables, not on the missing value itself. Men might be less likely to report their age. But among men, the missing ages aren't systematically different from the reported ones. This can still be safely imputed, provided the imputation model has access to the variables that predict missingness.
- **MNAR (Missing Not At Random):** missingness depends on the missing value itself: high earners decline to report income more often than everyone else. This is the dangerous case: naive imputation will systematically underestimate the very quantity being estimated. The mechanism that hides values correlates with those values. MNAR requires explicitly modelling the missingness process, not just filling gaps.

Given a mechanism, imputation strategies form a rough ladder of increasing complexity and (usually) increasing accuracy:

- **Drop** rows or columns outright.
- **Constant fill**: a fixed value such as 0, `"Unknown"`, or a sentinel.
- **Statistical fill**: the column's own mean, median, or mode (told you mode had a use).
- **Forward/backward fill**: for time series, where the adjacent value is a reasonable estimate.
- **K-nearest-neighbours imputation**: estimates a missing value from the most similar rows on other features.
- **Model-based imputation**: trains a regression or classifier to predict the missing column from the rest.

Climbing the ladder costs complexity and compute, and gets us robustness. Dropping is free but throws away information. KNN and model-based imputation are expensive but respect the structure between features.

7.2.2 Outliers

An outlier is a value far from the rest of a distribution. But "far" is not automatically "wrong." A CEO's salary in a company-wide compensation dataset is a real outlier. Treating it as noise discards signal. The question is always: *is this outlier informative or harmful?*

Two detection methods dominate practice. The **IQR (interquartile range)** method flags anything outside

$$[\,Q_1 - k \cdot \text{IQR},\quad Q_3 + k \cdot \text{IQR}\,], \qquad \text{IQR} = Q_3 - Q_1,\quad k = 1.5 \text{ (conventional)}$$

and is robust to skew because it's defined by rank statistics, not the mean.

> **What Is a Rank Statistic?**
>
> A rank statistic is computed from the sorted position of values, their rank, rather than their raw magnitude. Q_1 and Q_3 are examples: Q_1 is the value 25% of the way through the sorted data, Q_3 is the value 75% of the way through, and neither cares how far apart the surrounding values are. Move the largest value in a column from 100 to 100,000 and the mean shifts with it. Q_1 and Q_3 don't move at all, because their position in the sorted order hasn't changed. That's the same property Chapter 3 uses to call the median robust to outliers, applied here to quartiles instead of the midpoint.

The **z-score** method flags any value with

$$|z| > 3, \qquad z = \frac{x - \mu}{\sigma}$$

Here z is the number of standard deviations x sits from the mean μ, using the mean and standard deviation σ from Chapter 3. Unlike a rank statistic, z is computed from raw magnitude. That's exactly why it's simple and interpretable. But it assumes an approximately normal distribution. It's also somewhat circular: a genuine outlier inflates σ and can mask itself. Both are single-variable methods. Multivariate anomaly detection (Isolation Forest and its relatives, covered in Chapter 19) is needed when "unusual" only shows up in the *combination* of features rather than any one alone.

Once detected, treatment is a judgement call:

- **Remove** a value only when it's clearly an error, an age of 300, for instance.
- **Cap** (winsorise) it: replace values beyond a percentile threshold with the threshold. Use this when the value is real but shouldn't be allowed to dominate a model fit on scale-sensitive algorithms.
- **Transform** (typically log) a skewed distribution, to compress its tail rather than discard information from it.
- **Flag and keep**: add a binary `is_outlier` column and let the model decide. This works well for tree-based models that can split on the flag directly.

7.2.3 Duplicates

Duplicate rows are quietly catastrophic for machine learning. They corrupt evaluation, not just training. If some fraction of the training set is duplicated into the test set, reported accuracy is inflated by testing on examples the model has effectively memorised. The model looks better than it is. That gap only surfaces in production. The check is one line: `df.duplicated().sum()`. The fix requires judgement. Are the duplicates *exact*, every column matching? Then dropping is safe. Are they *partial*, same ID but a different timestamp (a re-submission or an update)? Then the right row to keep, first, last, or the one with the most complete data, depends on what the ID represents.

> **i** Memorisation at Internet Scale
>
> The same failure mode shows up at a much larger scale with generative AI. Internet-scale pretraining corpora overlap heavily with the questions people later ask these models. A language model's strong performance on a question is partly recall of material it has already seen, not proof of general reasoning. Performance measurably drops on questions that fall outside that training distribution. Book 2, Chapter 29 covers how transformer models are trained on corpora at this scale, and revisits this memorisation question directly.

7.2.4 Type Coercion

pandas infers each column's type from its values. One stray entry, such as `'unknown'` in a number column, makes the whole column load as text. Dates load as text unless pandas is told to parse them. Operating on such a column as text instead of numbers is a quiet, persistent source of bugs: string comparison orders lexically, so `'10' > '9'` evaluates to `False` (because `'1' < '9'` character-by-character). The numeric comparison `10 > 9` is `True`. The fix is to validate types immediately after loading data, before any filtering or aggregation touches it. Type errors compound: a wrong-typed column silently produces wrong-typed derived columns.

7.2.5 Encoding Errors

Text has to be stored as bytes. Turning bytes back into characters means picking a character encoding: UTF-8, Latin-1 (ISO-8859-1), Windows-1252, and a handful of others are all still common. Read a file with the wrong one and the bytes don't change, but the characters they turn back into do. A city name saved as UTF-8 and mistakenly read back as Latin-1 turns `São Paulo` into `SÃ£o Paulo`: the same underlying bytes, decoded through the wrong lookup table. This kind of corruption, known as mojibake, is silent. Nothing crashes. The

column still loads, still looks like a string, and one real category quietly splits into two, `São Paulo` and `SÃ£o Paulo` sitting side by side as if they were different cities.

Plain ASCII text (letters, digits, common punctuation) survives a wrong-encoding read unharmed. That's exactly why encoding errors are easy to miss: they only show up on the accented or non-Latin characters a dataset happens to contain. A column of English-only names can hide the same bug a column of international city names exposes immediately. Detection is mostly a matter of looking: scan `value_counts()` for a suspicious repeated pattern like `Ã` followed by another character, or count non-ASCII bytes directly. The fix, when the corruption follows the common UTF-8-read-as-Latin-1 pattern, is to run the mismatch backwards: re-encode the mangled string as Latin-1 to recover the original bytes, then decode those bytes as UTF-8 to get the real text back. When that round trip doesn't fully explain the corruption, re-reading the source file with the correct `encoding=` argument from the start is the safer fix.

7.2.6 Inconsistent Categories

Free-text categorical data is almost always inconsistent. It was typed by different people, systems, or eras. A gender column might contain `'female'`, `'Female'`, `'F'`, `'woman'`, and `'FEMALE'`: five spellings of the same underlying category. But to any model that one-hot encodes or otherwise treats string values literally, they're five *distinct* categories. The fix is aggressive normalisation: strip whitespace and lowercase as a floor, then explicitly map every observed variant to a small set of canonical values. And check what's left unmapped. An unmapped value silently becomes missing.

7.2.7 Schema Violations

A schema violation is a value that has the right type but breaks the rule that type is supposed to follow. `'not a date'` in a date column is not a type error. Python is perfectly happy holding it as a string. It's a schema violation: the column's whole reason for existing is to hold valid dates, and this value doesn't. The same pattern shows up as a negative age, a percentage over 100, an email address with no `@`, or a customer ID that points to a customer who was never created. Each of those parses fine and fails silently at the level that matters: meaning, not type.

Type Coercion (above) and schema violations often get caught by the same code, since forcing a column through `pd.to_datetime()` or `pd.to_numeric()` with `errors='coerce'` turns anything that isn't a date, or isn't a number, into a missing value automatically. But the two problems aren't the same thing. Type coercion asks *can this be parsed as the right type at all.* A schema violation can pass that test and still be wrong: age `-5` parses as a perfectly good integer. Catching that kind needs an explicit rule about what the field is

allowed to mean, not just what it's allowed to look like. That's exactly what Procedure 7.6's validation rules are for.

7.3 Trade-offs and Limits

The decision of whether to drop or impute a column comes down to how much is missing and why: dropping rows is safe when missingness is under roughly 5% and MCAR. The information loss is small and the bias risk is near zero. Above that threshold, or when the missing column matters to the outcome, imputation preserves far more of the dataset at the cost of introducing an estimate where a real value should be. One rule with no exceptions: never impute the target variable (the label). *Doing so is label fabrication.* The model would be trained, in part, on predictions of itself.

The decision of whether to remove or cap an outlier follows similar logic. Remove only when the value is clearly an error. Cap when it might be genuine but shouldn't be allowed to dominate a distance- or gradient-sensitive model. Flag and keep when outliers are themselves predictive. In fraud, sensor-failure, and financial contexts, they often are. The one step in this whole chapter that is never optional is validation. After any cleaning operation, compare the before/after distribution of every column that was touched. A cleaning step that silently collapses meaningful variation (over-aggressive winsorising that flattens a bimodal distribution, for instance) produces a dataset that is *cleaner-looking* but worse than the original. The damage is now invisible.

7.4 Procedures

Procedure 7.1 (Find and Remove Duplicates). *When to use:* as the first cleaning step on any new dataset, before anything else touches it.

Listing 7.1 Detecting and removing duplicate rows, exact and partial

```
# How many exact duplicates?
df.duplicated().sum()

# Remove duplicates (keep first occurrence)
df = df.drop_duplicates()
df = df.reset_index(drop=True)

# Deduplicate on a subset of columns
# (e.g. same ID, different timestamps)
df = df.drop_duplicates(subset=['id'], keep='last')
```

Common mistake: dropping duplicates without `reset_index(drop=True)` afterwards. The index still has gaps from the removed rows. Later `.loc[]` lookups that assume labels run from 0 to n−1 then miss rows or pick the wrong ones.

Procedure 7.2 (Cast Column Types). *When to use:* immediately after loading data, before any analysis or filtering. Never after. A wrong type propagates into every downstream computation.

Listing 7.2 Coercing text columns to their true numeric, datetime, and categorical types

```
# Numeric (coerce converts unparseable values to NaN
# rather than raising)
df['col'] = pd.to_numeric(df['col'], errors='coerce')

# Datetime -- explicit format is both faster and safer
# than inference
df['date'] = pd.to_datetime(
    df['date'], format='%Y-%m-%d', errors='coerce'
)

# Category (saves memory, enables category-aware operations)
df['col'] = df['col'].astype('category')
```

Procedure 7.3 (Normalise String and Category Columns). *When to use:* any free-text category column sourced from user input or a merge of multiple systems.

Listing 7.3 Normalising inconsistent category spellings to a canonical set

```
# Step 1: strip whitespace, lowercase
df['col'] = df['col'].str.strip().str.lower()

# Step 2: map every observed variant to a canonical value
mapping = {
    'male': 'male', 'm': 'male', 'man': 'male',
    'female': 'female', 'f': 'female', 'woman': 'female',
}
mapped = df['col'].map(mapping)

# Anything that didn't match the mapping becomes NaN --
# check the original values before overwriting them
unmapped = df.loc[mapped.isnull(), 'col'].value_counts()
df['col'] = mapped
```

Common mistake: applying the mapping without first checking `unmapped`. A variant that wasn't anticipated (`'Mal'`, a typo) silently becomes missing data instead of raising a visible error.

Procedure 7.4 (Detect and Handle Outliers). *When to use:* before imputation and feature engineering. Outliers distort the mean and median that imputation fills gaps with.

Listing 7.4 IQR-based outlier detection with two treatment options

```
def iqr_bounds(series, factor=1.5):
    q1, q3 = series.quantile(0.25), series.quantile(0.75)
    iqr = q3 - q1
    return q1 - factor * iqr, q3 + factor * iqr

lower, upper = iqr_bounds(df['col'])
outliers = df[(df['col'] < lower) | (df['col'] > upper)]

# Treatment A: winsorise (cap) -- keeps the row,
# limits its influence
df['col'] = df['col'].clip(lower=lower, upper=upper)

# Treatment B: flag and let the model decide
df['col_is_outlier'] = (
    (df['col'] < lower) | (df['col'] > upper)
).astype(int)
```

Procedure 7.5 (Impute Missing Values). *When to use:* after deduplication and type coercion, so the imputer sees clean numeric columns rather than text.

Listing 7.5 Imputation strategies ordered from simplest to most structure-aware

```
from sklearn.impute import KNNImputer

# Median imputation -- robust to skew and outliers, good
# default for numeric columns
df['col'] = df['col'].fillna(df['col'].median())

# KNN imputation -- estimates a missing value from the k
# most similar rows
imputer = KNNImputer(n_neighbors=5)
df[['col_a', 'col_b']] = imputer.fit_transform(
    df[['col_a', 'col_b']]
)
```

Common mistake: fitting the imputer on the full dataset, including the test set. This leaks test-set statistics into training. Fit only on the training split, then `.transform()` the test split with that same fitted imputer.

Procedure 7.6 (Validate a Cleaned Dataset). *When to use:* after every cleaning step, and always as the final gate before a cleaned dataset is handed to modelling.

Listing 7.6 A small rule-based validator for a cleaned dataset

```
def validate_dataset(df, rules):
    all_passed = True
    for check, description in rules:
        passed = check(df)
        print(f"{'OK' if passed else 'FAILED'}  {description}")
        all_passed = all_passed and passed
    return all_passed

rules = [
    (lambda d: d.isnull().sum().sum() == 0, 'No missing values'),
    (lambda d: d.duplicated().sum() == 0,    'No duplicate rows'),
    (lambda d: (d['age'] >= 16).all(),       'All ages >= 16'),
    (
        lambda d: (d['salary'] > 0).all(),
        'All salaries positive',
    ),
]
validate_dataset(df, rules)
```

Procedure 7.7 (Wrap Cleaning Steps in a Pipeline and ColumnTransformer). *When to use:* once the cleaning steps above are validated and stable. It turns them into a single reusable object and closes off leakage by construction, not by discipline alone.

Listing 7.7 Column-specific cleaning steps combined into one fit/transform object

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.preprocessing import OneHotEncoder

numeric_cols = ['age', 'salary', 'years_exp']
categorical_cols = ['gender']
```

```
numeric_pipeline = Pipeline([
    ('impute', KNNImputer(n_neighbors=5)),
])
categorical_pipeline = Pipeline([
    ('impute', SimpleImputer(strategy='most_frequent')),
    ('encode', OneHotEncoder(handle_unknown='ignore')),
])

preprocessor = ColumnTransformer([
    ('num', numeric_pipeline, numeric_cols),
    ('cat', categorical_pipeline, categorical_cols),
])

# Fit only on the training split
X_train_clean = preprocessor.fit_transform(X_train)
# Test data only ever gets transformed, never fit
X_test_clean = preprocessor.transform(X_test)
```

Common mistake: calling `.fit_transform()` on the test set too, "to keep it consistent with training." That re-introduces the exact leakage Procedure 7.5 already warned against, just one level up. The `ColumnTransformer` only prevents leakage by construction if it's fit once, on the training split, and every other split only ever calls `.transform()`.

7.5 Worked Example

The tutorial notebook behind this chapter builds its own mess deliberately. It generates a synthetic 500-employee dataset, then poisons it with every problem category from this chapter. That way the exact set of defects is known in advance. The cleaning pipeline can be checked against ground truth. Five exact-duplicate rows are appended and the whole set shuffled, producing a raw shape of **(505, 8)**.

Listing 7.8 Deliberately poisoning a clean synthetic dataset with every defect category

```
ages = np.random.randint(22, 65, n).astype(float)
missing_age_idx = np.random.choice(n, 40, replace=False)
ages[missing_age_idx] = np.nan
ages[0] = 312  # typo outlier

salaries = np.random.normal(70_000, 15_000, n)
salaries = np.clip(salaries, 30_000, 200_000)
# Data entry errors
salaries[np.random.choice(n, 5, replace=False)] = 999_999
```

```
missing_sal_idx = np.random.choice(n, 25, replace=False)
salaries[missing_sal_idx] = np.nan

# Wrong type
experience_str = [str(e) for e in np.random.randint(0, 30, n)]

# Encoding errors: UTF-8 city names read back as Latin-1
office_cities_messy = list(office_cities)
for i in np.random.choice(n, 8, replace=False):
    city = office_cities_messy[i]
    office_cities_messy[i] = city.encode('utf-8').decode('latin1')
```

The diagnostic pass (Procedure 7.1's `.duplicated()` plus `.isnull().sum()` and `.dtypes`) finds exactly what was planted. `age` is missing in **39 of 505 rows (7.7%)**: 40 were blanked, but the 312 typo then overwrote one of them. `salary` is missing in **25 rows (5.0%)**. There are **5 exact duplicate rows**. `years_exp` loaded as text instead of a number. And the raw `age` column's maximum is **312**, an obvious typo outlier no legitimate employee could have. Gender arrives split across ten spellings of two real categories (`'M'`, `'male'`, `'Male'`, `'MALE'`, `'man'` and their female equivalents). Two `hire_date` entries are schema violations, syntactically fine strings that aren't valid dates (`'not a date'` and an empty string). And `office_city` carries a real encoding error: 8 rows had their city name corrupted with a UTF-8-read-as-Latin-1 mismatch, and 7 of the 8 show up visibly mangled (`São Paulo` as `SÃ£o Paulo`, and so on). The eighth happened to land on a plain-ASCII city name, which the corruption passes straight through unchanged, invisible until it's understood *why*.

Cleaning proceeds in the order this chapter argues for, duplicates first, then types, then categories, then encoding, then outliers, then missingness, then validation:

Listing 7.9 The cleaning pipeline applied in dependency order

```
# 505 -> 500 rows
df = df_raw.drop_duplicates().reset_index(drop=True)

df['years_exp'] = pd.to_numeric(df['years_exp'], errors='coerce')
# 2 -> NaT
df['hire_date'] = pd.to_datetime(df['hire_date'], errors='coerce')

df['gender'] = (
    df['gender'].str.strip().str.lower().map(gender_map)
)

# Reverse the encoding mismatch: 7 mangled entries repaired.
# Correct values such as 'São Paulo' can't round-trip, so skip them.
def fix_mojibake(s):
```

```
    try:
        return s.encode('latin1').decode('utf-8')
    except (UnicodeDecodeError, UnicodeEncodeError):
        return s

df['office_city'] = df['office_city'].apply(fix_mojibake)

# 312 -> 90
df['age'] = df['age'].clip(lower=16, upper=90)
p01, p99 = df['salary'].quantile([0.01, 0.99])
# 999999 -> 112869
df['salary'] = df['salary'].clip(p01, p99)

# median = 42
df['age'] = df['age'].fillna(df['age'].median())
# KNN fills salary, using age and years_exp as neighbours
imputer = KNNImputer(n_neighbors=5)
impute_cols = ['age', 'salary', 'years_exp']
df['salary'] = imputer.fit_transform(df[impute_cols])[:, 1]
# Drops 2 NaT rows
df = df.dropna(subset=['hire_date'])
```

Running that pipeline on the actual generated data, deduplication takes the row count from 505 to exactly 500. The encoding repair recovers all 7 visibly mangled `office_city` entries back into their correct categories: `São Paulo` goes from 103 correctly-spelled plus 5 mangled to 108 correctly-spelled, and the same pattern repairs `Montréal` and `Zürich`. That confirms the UTF-8/Latin-1 round trip was the right diagnosis, not a guess. Domain-knowledge capping brings the age maximum down from 312 to 90. The IQR method alone flags only 1 outlier in the age column. That one outlier is the 312: IQR's upper fence sits at 84.5. But IQR only flags values, and its fence moves with the data. The domain-knowledge cap of `[16, 90]` states what a valid age is, so it is applied on top of IQR rather than instead of it. Winsorising salary at the 1st/99th percentile takes the range from **$30,000–$999,999** down to **$33,061–$112,869**. The five $999,999 entry errors are pulled all the way down to the 99th-percentile threshold. Gender normalisation collapses the ten observed spellings into exactly two canonical categories: **267 male, 233 female**. After median-imputing age (median = **42**), KNN-imputing salary, and dropping the 2 rows with unrecoverable hire dates, the final validated dataset has **498 rows, zero missing values, and zero duplicates**. That's down from 505 raw rows, a loss of under 1.5%, with every remaining value inside a safe domain range and every city name spelled the same way twice.

7.6 History and Further Reading

Murphy's introductory volume gives the formal treatment of data distributions and their properties that underlies the MCAR/MAR/MNAR distinction in this chapter (Murphy 2022). Burkov's discussion of feature engineering assumes clean input arrives first. That's precisely the boundary this chapter defends (Burkov 2019). Thakur's practical chapters on encoding and feature strategies pick up exactly where a cleaned dataset like this chapter's leaves off. Worth reading immediately after this chapter, not before (Thakur 2020). Géron's *Hands-On Machine Learning*, Ch. 2, builds a full Scikit-Learn cleaning pipeline (imputation, custom transformers, encoding) end to end. It's a good template for turning this chapter's individual techniques into a single reusable pipeline (Géron 2019).

7.7 How This Connects

This chapter builds on the pandas mechanics of Chapter 6 and feeds directly into the exploratory data analysis of Chapter 9. In practice the two are often interleaved. EDA is often where a data quality problem is first noticed. It enables every modelling chapter that follows: "garbage in, garbage out" is a cliché because it keeps being true. It is often confused with feature engineering (Chapter 14). Cleaning removes errors and standardises what's already there. Engineering creates new signal from it. Cleaning has to come first. Engineered features built on dirty inputs propagate the dirt into new columns.

7.8 Exercises

1. [**W**] Classify each of the following as MCAR, MAR, or MNAR, and justify your answer in one sentence: (a) a temperature sensor that randomly drops 2% of readings due to a hardware fault; (b) a salary survey where respondents earning over $500k are half as likely to answer the income question as everyone else; (c) a customer age field where missingness is higher among users who signed up via a mobile app that didn't require an age field, regardless of their actual age.
2. [**W**] Using Listing 7.8's logic, explain why `ages[0] = 312` is assigned *after* the missing-value assignment `ages[missing_age_idx] = np.nan`. Explain what would change about row 0 if the two lines were swapped.
3. [**A**] Using Listing 7.9 as a reference, compute what the age imputation value would need to be if, instead of median imputation, mean imputation were used on the post-capping age column. Describe the one-line change. State whether you'd expect the result to be higher or lower than 42, and why.

4. [**A**] The chapter reports that IQR-based detection flags only 1 outlier in the raw age column despite there being an obvious typo (312). Using Listing 9.6's `iqr_bounds` function, explain why a single extreme value like 312 adds only one to the outlier count and barely moves the fence, given how Q_1 and Q_3 are computed.
5. [**E**] A colleague proposes skipping validation (Procedure 7.6) on the grounds that "the pipeline already ran without errors, so the data must be fine." Using this chapter's Trade-offs section, write the two or three sentences you'd say to push back, with a concrete example of a silent failure that would pass unnoticed without an explicit validation step.
6. [**E**] Design a validation rule set (in the style of Listing 7.6) for a dataset of e-commerce orders with columns `order_total`, `discount_pct`, `ship_date`, `delivery_date`, and `customer_id`. Propose at least four rules, including at least one that checks a relationship *between* two columns rather than a single column in isolation.
7. [**R**] MNAR missingness is described in this chapter as requiring you to "model the missingness mechanism, not just the missing value." Research one concrete technique for this (e.g. Heckman correction, pattern-mixture models, or multiple imputation with a missingness indicator) and summarise, in your own words, what data beyond the column itself it requires.

7.9 Mental Model

Data cleaning is defensive programming for a dataset: every assumption you don't check is a bug waiting to surface in a model's predictions. The goal is not a perfect dataset (real data will never get there). It's a dataset where every anomaly has been noticed, understood, and deliberately handled, rather than silently ignored. The pipeline in this chapter's Worked Example turned 505 raw rows with seven planted defect categories into 498 clean rows and zero silent errors. That ratio is the realistic target.

8 SQL for Data Science

Most Solutions Architects already know how to write `SELECT * FROM orders WHERE status = 'active'`. That's operational SQL. Retrieving specific records. What separates it from the SQL a data scientist writes is intent. Analytical SQL summarises, ranks, compares across time windows, and answers questions like "which customers increased their spend the most last quarter?" Most business data lives in relational databases or warehouses rather than tidy CSV files. Fluency in this analytical style is a prerequisite for working with real data. This chapter treats SQL as a language for asking questions of *sets* of rows, rather than a language for retrieving individual ones. It builds up from `GROUP BY` to the two tools that make that shift possible: common table expressions and window functions.

8.1 Motivation

The gap between "I can write a `WHERE` clause" and "I can write the query that answers the business question" is the gap between operational and analytical SQL. It's where a lot of otherwise-capable engineers stall the first time they're handed a warehouse table and a business question instead of a well-specified lookup. Relational databases store data in tables. SQL performs *set operations* on those tables: filtering, joining, grouping, transforming collections of rows at once. Declaratively. You describe what you want and the query planner decides how. The shift that unlocks analytical SQL is thinking in terms of **aggregations over groups**. Almost every analytical question reduces to "for each *group*, what is the *aggregate* of *value*?" That's what `GROUP BY` plus an aggregate function expresses, and what the rest of this chapter's tools extend.

8.2 Exposition

8.2.1 GROUP BY and Aggregations

`GROUP BY` partitions the rows of a table into groups sharing a common value, then collapses each group down to a single summary row:

Listing 8.1 Grouped aggregation: headcount and payroll by department

```
SELECT
    department,
    COUNT(*)            AS headcount,
    AVG(salary)         AS avg_salary,
    MAX(salary)         AS max_salary,
    SUM(salary)         AS total_payroll
FROM employees
GROUP BY department
ORDER BY total_payroll DESC;
```

`HAVING` is the group-level equivalent of `WHERE`. It filters *after* the grouping and aggregation have happened. That's why it can reference an aggregate expression that doesn't exist until the grouping is done:

Listing 8.2 HAVING filters groups, not rows

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
-- Only departments with high average salary
HAVING AVG(salary) > 80000;
```

`WHERE` and `HAVING` don't do the same job. `WHERE` restricts which rows enter a group in the first place. `HAVING` restricts which already-computed groups survive into the output.

8.2.2 CTEs — Common Table Expressions

A **common table expression (CTE)** is a named subquery defined at the top of a query with `WITH`. Its purpose is readability, not performance. It lets a complex query be built up as a sequence of named, single-purpose steps, the same way a complicated Python function gets broken into smaller named helper functions, rather than one dense expression.

Listing 8.3 A two-step CTE: monthly revenue, then growth against the prior month

```
WITH monthly_revenue AS (
    SELECT
        DATE_TRUNC('month', order_date) AS month,
        SUM(amount)                     AS revenue
    FROM orders
    GROUP BY 1
),
revenue_with_growth AS (
    SELECT
        month,
        revenue,
        LAG(revenue) OVER (ORDER BY month)
            AS prev_revenue,
        revenue - LAG(revenue) OVER (ORDER BY month)
            AS growth
    FROM monthly_revenue
)
SELECT * FROM revenue_with_growth WHERE month >= '2023-01-01';
```

A useful discipline: each CTE should do exactly one thing. Its name should describe that thing well enough that the final `SELECT` reads almost like English.

8.2.3 Window Functions — The Most Powerful Analytical Tool

A **window function** computes a value across a set of rows *related to the current row*, without collapsing those rows the way `GROUP BY` does. This is the distinction: `GROUP BY` produces one row per group. A window function keeps every original row and adds a computed column alongside it. The general shape is:

Listing 8.4 Window function syntax: PARTITION BY, ORDER BY, and an optional frame

```
function_name() OVER (
    -- like GROUP BY, but doesn't collapse rows
    PARTITION BY grouping_column
    -- defines the sequence for ordered functions
    ORDER BY ordering_column
    -- optional: defines the window frame
    ROWS BETWEEN ...
)
```

Three families cover most analytical needs. **Ranking functions**: `ROW_NUMBER()` assigns 1, 2, 3… with no ties. `RANK()` assigns 1, 1, 3…, where tied rows share a rank and a gap follows. `DENSE_RANK()` assigns 1, 1, 2…, with no gap. **Offset functions**: `LAG(col, n)` reaches back `n` rows, `LEAD(col, n)` reaches forward, both within the current partition and ordering. **Aggregate window functions**: the same `SUM`, `AVG`, `COUNT` used with `GROUP BY`, but applied `OVER` a window instead. `SUM(amount) OVER (ORDER BY date)` is a running total. With `ORDER BY` and no frame clause, the default frame is `RANGE`: rows tied on the ordering column share one running total. Write `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` to step row by row. A `ROWS` frame counts rows, not days, so a 7-row moving average covers 7 days only when every day has exactly one row. `COUNT(*) OVER ()`, an empty `OVER ()`, attaches the grand total row count to every row without filtering or grouping anything.

8.2.4 Joins for Analysis

Most analytical questions pull data from more than one table. `INNER JOIN` keeps only rows matching in both tables. `LEFT JOIN` keeps every row from the left table, with `NULL`s filled in where the right table has no match. `RIGHT JOIN` is the mirror image; rare in practice, since a `LEFT JOIN` with the tables reordered reads more naturally. `FULL OUTER JOIN` keeps every row from both sides. In analytics, `LEFT JOIN` dominates: there is almost always a *fact* table (orders, events, transactions) that needs enriching with *dimension* data (customer or product attributes) while every fact row is preserved regardless of whether a match exists.

8.2.5 Using SQL at Scale

A query that runs fine on a thousand-row table can take minutes, or fail outright, on a billion-row one. The SQL hasn't changed. What has changed is how much data the engine has to touch to answer it. A badly formed query touches far more of it than necessary.

The starting point for any slow query is `EXPLAIN`, or `EXPLAIN ANALYZE`, which runs the query and reports real timings instead of estimates. It shows the query plan: the sequence of scans, joins, and aggregations the engine chose, and roughly how much work each step costs. A **sequential scan** reads every row in a table. An **index scan** reads only the rows that match a condition, using a structure built in advance to look them up directly. On a large table, the difference between the two is often the difference between milliseconds and minutes.

Four habits keep a query fast as data grows:

- **Filter early.** A `WHERE` clause applied before a `JOIN` shrinks the table the join has to process. Most modern query planners reorder this automatically, but not always. A query that filters manually inside a CTE before joining never depends on the planner getting it right.
- **Select only the columns needed.** `SELECT *` costs little on a small row-based table. On the columnar formats warehouses use, Parquet, Redshift, Snowflake, each column is stored separately. A wide `SELECT *` on a hundred-column table forces the engine to read all hundred columns off disk, even when a query only needs three of them.
- **Avoid wrapping a filtered column in a function.** `WHERE YEAR(order_date) = 2023` cannot use an index on `order_date`, because the engine would have to evaluate `YEAR()` on every row before it can tell which ones qualify. `WHERE order_date >= '2023-01-01' AND order_date < '2024-01-01'` expresses the identical filter in a form the engine can push down to an index or a partition boundary.
- **Know what's driving the join.** A join between a small table and a large one is cheap if the engine can broadcast the small table to every worker. A join between two large tables usually requires shuffling data across the network, the single most expensive operation in a distributed query engine. Filtering both sides down before the join, not after, is what keeps that shuffle small.

At warehouse scale, these habits are not just about latency. Tools like Amazon Athena bill per byte scanned: a query that reads every column of every partition costs measurably more than the same query with column pruning and partition filters applied, independent of how long it takes to run.

8.3 Trade-offs and Limits

SQL and pandas overlap enough that the real skill is knowing which to reach for, not picking one permanently. SQL is the right tool when the data already lives in a database or warehouse, when a query joins more than two tables, or when the aggregation runs over more rows than fit comfortably in memory. A well-indexed warehouse table can aggregate billions of rows in a way a laptop's RAM cannot. pandas is the right tool when the data is already loaded as a DataFrame, when the transformation is a custom one specific to feature engineering for a model, or when the task is quick exploration of a CSV that was never going to touch a database. The strongest data scientists are fluent in both and treat them as adjacent stages of a pipeline rather than competitors. SQL does the aggregation in the database, where it's fast and where the data already lives. pandas does the final feature engineering in Python, where custom logic and ML libraries live. Trying to force one tool to do the other's job (writing a ten-table join as chained pandas `merge()` calls, or trying to hand-roll a gradient-boosted feature transform in a `CASE WHEN`) is an avoidable source of slow, unreadable code.

8.4 Procedures

Procedure 8.1 (Register a DataFrame as a SQL Table and Query It). *When to use:* data is already in Python as a pandas DataFrame, and the next transformation is more naturally expressed as SQL than as chained pandas calls. This is most often the case once a query needs a `GROUP BY`, a multi-table `JOIN`, or a window function.

Listing 8.5 DuckDB: registering a DataFrame as a queryable SQL table

```
import duckdb

con = duckdb.connect()
con.register('my_table', df)  # register DataFrame as a SQL table

# Run a query and get a DataFrame back
result = con.execute(
    "SELECT * FROM my_table WHERE value > 100"
).df()

# Shorthand helper used throughout this chapter
def sql(query): return con.execute(query).df()

# DuckDB can also query DataFrames directly by variable name
result = duckdb.sql("SELECT * FROM df WHERE col > 5").df()
```

Common mistake: relying on DuckDB finding a DataFrame by its variable name. It does, through both `duckdb.sql(...)` and a connection's `execute()`, but only while that variable is in scope. Query it from somewhere else and DuckDB raises `CatalogException`. `con.register('name', df)` gives the table a fixed name that works anywhere the connection is used.

Procedure 8.2 (Filter Rows with `pandas.query` as a Lightweight Alternative). *When to use:* the only operation needed is row filtering and setting up a DuckDB connection would be overkill.

Listing 8.6 pandas.query for simple filtering without a SQL engine

```
result = df.query("age > 30 and salary < 80000")

# Reference a Python variable in the query string with @
threshold = 30
result = df.query("age > @threshold")

result = df.query("department == 'Engineering'")
```

Common mistake: reaching for `pandas.query` on a task that needs a `GROUP BY` or a window function. It supports row filtering only, nothing else. The moment a query needs an aggregation across groups, switch to Procedure 8.1.

Procedure 8.3 (Aggregate with GROUP BY and HAVING). *When to use:* the business question is a "for each X, what is the Y" question, the single most common shape in analytical work.

Listing 8.7 GROUP BY with a HAVING filter and a conditional aggregation

```
SELECT col, COUNT(*), SUM(val), AVG(val), MIN(val), MAX(val)
FROM table
GROUP BY col
HAVING SUM(val) > 1000
ORDER BY SUM(val) DESC;

-- Conditional aggregation: split one column's sum by a condition
SELECT
    SUM(CASE WHEN status = 'completed'
             THEN revenue ELSE 0 END) AS completed_rev,
    SUM(CASE WHEN status = 'returned'
             THEN revenue ELSE 0 END) AS returned_rev
FROM orders;
```

Common mistake: referencing a column *alias* inside `HAVING` (e.g. `HAVING avg_salary > 80000` when `avg_salary` was defined with `AS` in the `SELECT`). Standard SQL evaluates `HAVING` before aliases are resolved, and PostgreSQL, SQL Server and Oracle reject the alias. DuckDB, MySQL and SQLite accept it. Repeating the full aggregate expression works everywhere.

Procedure 8.4 (Build Multi-Step Logic with CTEs). *When to use:* any query with more than two logical steps. Past that point, a single nested query becomes unreadable and hard to debug. A chain of named CTEs is not.

Listing 8.8 The general CTE shape: named steps, each building on the last

```
WITH
cte_name_1 AS (
    SELECT ...      -- step 1
),
cte_name_2 AS (
    SELECT ...      -- step 2, may reference cte_name_1
    FROM cte_name_1
    WHERE ...
),
cte_name_3 AS (
    SELECT
        *,
        RANK() OVER (PARTITION BY ... ORDER BY ...) AS rank
    FROM cte_name_2
)
SELECT * FROM cte_name_3
WHERE rank <= 10;
```

Common mistake: treating a CTE referenced multiple times in the same query as if it were materialised once. Some engines re-evaluate it at each reference. Others, such as PostgreSQL 12 and later, materialise it. Check the query plan. If a CTE is expensive and reused, a temp table makes the choice explicit.

Procedure 8.5 (Rank and Compare Rows with Window Functions). *When to use:* a per-row computed value is needed: a rank, a running total, a comparison to the previous row, without collapsing the rows the way `GROUP BY` would.

Listing 8.9 Window functions: running total, previous-row lookup, in-group rank, and a moving average

```
SELECT
    customer_id,
    order_date,
    revenue,

    -- Running total (cumulative sum)
    SUM(revenue) OVER (
        PARTITION BY customer_id ORDER BY order_date
    ) AS running_total,

    -- Previous row's value
```

```
    LAG(revenue, 1) OVER (
        PARTITION BY customer_id ORDER BY order_date
    ) AS prev_revenue,

    -- Rank within a group
    RANK() OVER (
        PARTITION BY region ORDER BY revenue DESC
    ) AS rank_in_region,

    -- Moving average (last 7 rows)
    AVG(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS moving_avg_7

FROM orders;
```

Common mistake: expecting `LAG()` to return zero for the first row in a partition. It returns `NULL`, because there is no previous row. Wrap it in `COALESCE(LAG(col) OVER (...), 0)` if a zero is what the downstream calculation needs.

Procedure 8.6 (Compute Period-over-Period Growth). *When to use:* month-over-month, quarter-over-quarter, or year-over-year growth calculations: a `GROUP BY` to get one row per period, followed by `LAG` to compare consecutive periods.

Listing 8.10 Period-over-period growth: aggregate to a period, then compare with LAG

```
WITH monthly AS (
    SELECT
        DATE_TRUNC('month', order_date) AS month,
        SUM(revenue) AS revenue
    FROM orders
    GROUP BY 1
)
SELECT
    month,
    revenue,
    LAG(revenue) OVER (ORDER BY month) AS prev_month,
    revenue - LAG(revenue) OVER (ORDER BY month) AS abs_change,
    (revenue / LAG(revenue) OVER (ORDER BY month) - 1) * 100
        AS pct_change
FROM monthly;
```

Common mistake: forgetting that DuckDB's date arithmetic (`INTERVAL '7 days'`) is not portable. SQLite, for instance, expects `date(col, '+7 days')`. Check the specific engine's date-function syntax before assuming a query will run unmodified elsewhere.

Procedure 8.7 (Diagnose a Slow Query with EXPLAIN). *When to use:* a query is slower than expected and the reason isn't obvious from reading the SQL alone.

Listing 8.11 Reading a query plan with EXPLAIN ANALYZE

```
EXPLAIN ANALYZE
SELECT c.customer_id, SUM(o.revenue) AS ltv
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status = 'completed'
GROUP BY c.customer_id;
```

Common mistake: running plain `EXPLAIN` and reading its cost estimates as if they were real timings. Estimates come from table statistics that can be stale or wrong. `EXPLAIN ANALYZE` executes the query and reports real row counts and timings at each step, which is what a genuine performance investigation needs.

8.5 Worked Example

The tutorial notebook builds a small, realistic in-memory e-commerce database with DuckDB (200 customers, 50 products, and 3,000 orders, generated with `np.random.seed(42)` so every number below is exactly reproducible), then answers six business questions against it with increasingly complex SQL. Of the 3,000 orders, 2,529 (84.3%) are `completed`, 321 (10.7%) `returned`, and 150 (5.0%) `pending`. Every query below filters to `completed` orders. Returned or pending revenue hasn't landed yet.

The first question is a plain `GROUP BY`: which product category generates the most completed revenue?

Listing 8.12 Revenue by category: a straight application of Procedure 8.3

```
SELECT
    p.category,
    COUNT(o.order_id)          AS total_orders,
    ROUND(SUM(o.revenue), 2)   AS total_revenue,
    ROUND(AVG(o.revenue), 2)   AS avg_order_value
FROM orders o
JOIN products p ON o.product_id = p.product_id
WHERE o.status = 'completed'
GROUP BY p.category
ORDER BY total_revenue DESC;
```

Sports leads on total revenue ($630,002.73 across 756 orders), ahead of Clothing ($553,492.97, 535 orders). But Clothing has the highest *average* order value at $1,034.57, versus Sports' $833.34. That's easy to miss if the chart only shows the total-revenue bar. Home ($408,763.29), Electronics ($344,965.32), and Books ($179,920.02, the smallest category) round out the ranking. Total revenue and average order value answer different business questions. A `GROUP BY` computes both from the same rows for free. There's rarely a reason not to pull both.

The second question chains a CTE with a window function to answer something `GROUP BY` alone cannot: what fraction of total revenue comes from the top customers, the Pareto question?

Listing 8.13 Cumulative revenue share via a running total over a grand total: the Pareto curve

```
WITH customer_ltv AS (
    SELECT c.customer_id, ROUND(SUM(o.revenue), 2) AS ltv
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id
)
SELECT
    ROW_NUMBER() OVER (
        ORDER BY ltv DESC, customer_id
    ) AS customer_rank,
    ROUND(
        SUM(ltv) OVER (
            ORDER BY ltv DESC, customer_id
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

```
        )
        / SUM(ltv) OVER () * 100,
        2
    ) AS cum_pct
FROM customer_ltv;
```

The windowed `SUM(ltv)` is a running total ordered by descending value, with `customer_id` breaking ties. Dividing by `SUM(ltv) OVER ()`, the same aggregate with an empty window, i.e. the grand total attached to every row, converts that running total into a cumulative percentage without a second query. On this dataset it takes 138 of the 200 customers (69.0%) to reach 80% of total revenue, and 74 customers (37.0%) to reach 50%. That is a noticeably flatter curve than the classic 80/20 rule. That's worth pausing on. It's a direct consequence of how the data was generated: revenue per order is `quantity × price`, drawn from independent, roughly uniform distributions, not from the heavy-tailed distribution real purchasing behaviour usually follows. A synthetic dataset built from independent uniform draws will never produce as concentrated a Pareto curve as real transaction data does. That's the distributional reasoning from Chapter 3 at work.

The third question is the capstone: of customers whose *first* order fell in Q1 2022, what fraction placed a second order within 90 days?

Listing 8.14 Cohort retention: first order, cohort membership, and second order CTEs joined together

```
WITH first_orders AS (
    SELECT customer_id, MIN(order_date) AS first_order_date
    FROM orders
    WHERE status = 'completed'
    GROUP BY customer_id
),
q1_cohort AS (
    SELECT customer_id, first_order_date
    FROM first_orders
    WHERE first_order_date >= '2022-01-01'
      AND first_order_date <  '2022-04-01'
),
second_orders AS (
    SELECT
        o.customer_id,
        o.order_date,
        ROW_NUMBER() OVER (
            PARTITION BY o.customer_id ORDER BY o.order_date
        ) AS order_num
    FROM orders o
```

```
    JOIN q1_cohort c ON o.customer_id = c.customer_id
    WHERE o.status = 'completed'
      AND o.order_date > c.first_order_date
)
SELECT
    COUNT(DISTINCT q1_cohort.customer_id) AS cohort_size,
    COUNT(DISTINCT CASE
        WHEN second_orders.order_num = 1
         AND second_orders.order_date
             <= q1_cohort.first_order_date + INTERVAL '90 days'
        THEN second_orders.customer_id
    END) AS retained_90d
FROM q1_cohort
LEFT JOIN second_orders
    ON q1_cohort.customer_id = second_orders.customer_id;
```

The Q1 2022 cohort has 156 customers; 128 of them (82.1%) placed a second order within 90 days. That retention figure is high enough to be suspicious for a real e-commerce business. And it is, for the same reason the Pareto curve above was flatter than expected. Orders were assigned to customers uniformly at random across a 730-day window. So a customer with even a modest order count is, by construction, likely to have *some* order land in the next 90 days purely by chance. The lesson generalises past this dataset: a retention number computed correctly from synthetic data can still be structurally meaningless if the data-generating process doesn't resemble the real process it's standing in for. The query being right is necessary, not sufficient.

AWS Athena / Redshift Equivalent

```
-- These same queries run in Amazon Athena (S3-backed
-- serverless SQL) and Amazon Redshift (data warehouse)
-- with minimal changes.

-- Athena date truncation (same as DuckDB)
DATE_TRUNC('month', order_date)

-- Redshift window function (identical syntax)
SUM(revenue) OVER (PARTITION BY customer_id ORDER BY order_date)

-- Athena reads Parquet directly from S3:
-- CREATE EXTERNAL TABLE orders ...
-- LOCATION 's3://my-bucket/orders/'
```

When to prefer Athena over pandas + DuckDB: once the data lives in S3 and is too large to comfortably fit in memory. Athena scales horizontally and is billed per byte of data scanned. For prototyping on a sample or working set that fits in memory, pandas + DuckDB is faster to iterate with and free.

8.6 History and Further Reading

Thakur's practical approach applies pandas to many of the same operations covered in this chapter. It's worth reading side-by-side for the comparison (Thakur 2020). For a foundational, SQL-only reference that goes considerably deeper into syntax and query planning than this chapter needs to, Beaulieu's *Learning SQL* is the standard recommendation (Beaulieu 2005). DuckDB's own documentation (duckdb.org) is unusually readable for a database's docs. It's worth bookmarking directly, since it's the engine every listing in this chapter runs against. For how these same concepts map onto managed AWS services at scale, the Athena and Redshift documentation covers the translation referenced in this chapter's AWS callout.

8.7 How This Connects

This chapter builds on basic SQL (`SELECT`, `WHERE`, `JOIN`) together with Chapter 6's pandas material. The two are used side by side throughout the Worked Example, with SQL doing the aggregation and pandas or DuckDB moving results back and forth. It enables data pipeline construction (Book 2, Chapter 24) and feature engineering pulled directly from database sources (Chapter 14), both of which assume the `GROUP BY`/CTE/window-function vocabulary built here. It's most often confused with pandas, since the two overlap a lot in what they can express. But the right framing is not "which is better" but "where does the data live": SQL belongs in the database, pandas belongs in Python, and most real pipelines use both.

8.8 Exercises

1. **[W]** In one or two sentences, explain why `HAVING SUM(revenue) > 1000` works but `HAVING total_revenue > 1000` (referencing an alias defined as `SUM(revenue) AS total_revenue` in the same query) fails in many engines.
2. **[W]** Explain the difference in output between `RANK()`, `DENSE_RANK()`, and `ROW_NUMBER()` when three rows are tied for first place, using Listing 8.9's `rank_in_region` column as a starting point.
3. **[A]** Adapt Listing 8.12 to group by `region` instead of `category` (joining `orders` to `customers` rather than `products`). Which region has the highest total completed revenue, and how close is it to the second-place region?
4. **[A]** Using Listing 8.14 as a base, change the retention window from 90 days to 60 days. Does the retention rate change substantially, and does that change your reading of how meaningful the original 90-day figure was?

5. [**A**] Using Listing 8.13's output, determine how many customers are needed to reach 50% of total revenue, and express that as a percentage of the full 200-customer base.
6. [**E**] The Worked Example argues the dataset's Pareto curve is flatter than real business data because revenue was generated from independent uniform draws rather than a heavy-tailed distribution. Propose a concrete change to the data-generating code in the tutorial notebook (a specific distribution or dependency between variables) that would produce a sharper, more realistic 80/20 split. Explain why your change would have that effect.
7. [**E**] Write a query skeleton (not necessarily runnable against this chapter's schema) that uses conditional aggregation (`SUM(CASE WHEN ... THEN ... ELSE 0 END)`) to compute, in a single `GROUP BY category` query, both completed revenue and returned revenue side by side as two columns. Explain why this is preferable to running two separate queries and joining the results.
8. [**R**] Read DuckDB's documentation on the logical order of SQL clause evaluation (`FROM` → `WHERE` → `GROUP BY` → `HAVING` → window functions → `SELECT` → `ORDER BY`). Explain why window functions are evaluated after `WHERE`/`GROUP BY`/`HAVING` but before the final `SELECT` list is resolved. Explain what that ordering implies about why a window function can't appear directly inside a `WHERE` clause.

8.9 Mental Model

SQL is a language for asking questions of sets. Not describing procedures. Every analytical question worth writing a query for reduces to the same underlying operation. Partition the data into groups, and optionally order the rows within those groups. Then either aggregate the group down to one row (`GROUP BY`) or attach a computed value to every row without collapsing it (a window function). Once every query is read through that lens, window functions stop being a scary advanced feature and become the obvious tool for "I need `GROUP BY`'s aggregation, but I need to keep all my rows."

9 Exploratory Data Analysis & Visualisation

Most modelling failures are not algorithmic. They are about what the data scientist failed to notice. A data scientist ships a model without noticing the target is heavily imbalanced, that two features are nearly redundant, or that a column silently changed units after a database migration. Exploratory Data Analysis (EDA) is the discipline that catches these problems before they become modelling problems. A structured, repeatable way of understanding a dataset before any modelling decision is made. It is not a step performed once. It is a mindset that a data scientist returns to after every surprising result. Every new feature. Every model iteration. This chapter builds that mindset into a repeatable sequence: look at each variable alone, look at pairs of variables, look at everything together, and write down what surprised you.

9.1 Motivation

EDA is a structured conversation with a dataset. The data scientist asks questions and lets the data answer, in a natural progression:

- **What do I have?** Shape, types, nulls, duplicates.
- **What does each variable look like on its own?** Distributions, outliers, skew.
- **How do variables relate to each other?** Correlations, conditional distributions.
- **How do variables relate to the target?** The most modelling-relevant question.
- **What surprised me?** The important question, asked last on purpose: only once the first four are answered is there enough context to recognise a genuine surprise.

There is no fixed script for EDA. The path is determined by what the data shows at each step. A surprising distribution at the univariate stage sends the analysis back to the data source. An unexpected correlation at the bivariate stage generates a hypothesis worth testing before any model is trained.

9.2 Exposition

9.2.1 Univariate Analysis

Univariate analysis examines one variable in isolation. For numeric variables, four properties matter. **Central tendency**: mean and median. If they diverge, the distribution is skewed. **Spread**: standard deviation and interquartile range. A spread near zero means the variable is nearly constant and gives a model little to use. **Shape**: bell-shaped, skewed, bimodal, or uniform. **Outliers**: extreme values that could be data-entry errors, or unusual, informative cases. For categorical variables, the properties that matter are different. **Cardinality**: how many unique values. A free-text column with thousands of unique values needs different treatment than a five-level category. **Frequency distribution**: is one category dominant enough to make the variable nearly constant? **Rare categories**: values so infrequent that no model can learn a reliable pattern from them.

9.2.2 Bivariate Analysis

Bivariate analysis examines two variables together. The right chart depends on the types involved.

- **Two numeric variables.** A scatter plot shows the relationship shape directly. A correlation coefficient summarises its strength: Pearson correlation for linear relationships, Spearman correlation for monotonic-but-non-linear ones.
- **A numeric variable against a categorical one.** Box plots or violin plots show whether the numeric distribution shifts across categories. If it does, the categorical variable carries information.
- **Two categorical variables.** Cross-tabulations and stacked bar charts reveal whether knowing one tells you something about the other.

The most important bivariate analysis in a modelling context is **any variable against the target**: does this feature's distribution differ between target classes, or correlate with target values? If it doesn't, the feature is a weak candidate for the model regardless of what it measures, or how important the customer thinks it is.

9.2.3 Multivariate Analysis

Multivariate analysis looks at three or more variables simultaneously. A **pairplot** (seaborn's `pairplot`) draws a scatter plot for every pair of numeric variables at once, with each variable's own distribution on the diagonal, coloured by a categorical variable of interest. It's the fastest way to see a dataset's full correlation structure in one figure. A **correlation heatmap** shows the Pearson correlation between every pair of numeric variables at a glance:

values near ±1 flag strong linear relationships, values near 0 mean no linear relationship. Critically, a pair correlated above roughly 0.95 is close to redundant. Including both in a model rarely helps and sometimes hurts.

9.2.4 The EDA Mindset

The most valuable output of an EDA session is not a chart. It is a written **list of hypotheses and surprises**: things that didn't match expectations. Leads worth following up. Modelling decisions the data is suggesting. Writing these down as they arise, instead of trusting memory, is what turns EDA into a narrative that connects to the feature engineering decisions of Chapter 14.

9.3 Trade-offs and Limits

How much EDA is enough? Enough to answer these questions:

- Are there data quality issues that need cleaning?
- Which features look promising?
- Are there obvious feature engineering opportunities?
- Is the target variable behaving as expected?

Going further than that risks **data snooping**: learning so much about a dataset, including its eventual test set, that later feature choices are implicitly optimised for data the model should never have seen. Running full EDA on a complete dataset before any train/test split is a common mistake for exactly this reason: patterns discovered that way may not hold on new data.

A second trade-off is visualisation versus summary statistics. Charts reveal shapes that statistics can hide entirely. The classic example is Anscombe's quartet, four datasets with identical means, variances, and correlation coefficients but visibly different shapes when plotted. The rule is to visualise, even when the summary numbers look informative. A boxplot alone, for instance, can make a bimodal distribution look identical to a unimodal one. Boxplots show quartiles, not shape.

The same trick is easy to reproduce with fresh data. That makes the point more easily than quoting Anscombe's original numbers: construct four datasets from scratch, one point at a time, so that all four end up with the same mean, spread, and correlation.

Listing 9.1 Four datasets built by hand to share the same mean, spread, and correlation, but not the same shape

```
import numpy as np

rng = np.random.default_rng(11)
x = np.array([2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 20], dtype=float)

# A: a genuine linear relationship, plus noise
y_linear = 0.60 * x + 2.6 + rng.normal(0, 2.9, len(x))
target_mean, target_std = y_linear.mean(), y_linear.std(ddof=1)

def match_stats(y_raw, mean, std):
    """Rescale y_raw to a target mean and std. An affine rescale like
    this changes nothing about the shape, or the correlation with x."""
    z = (y_raw - y_raw.mean()) / y_raw.std(ddof=1)
    return z * std + mean

# B: a curve, not a line
y_curved_raw = -0.0644 * (x - 13.97) ** 2 + 40 + rng.normal(0, 0.5, len(x))
y_curved = match_stats(y_curved_raw, target_mean, target_std)

# C: the same line as A, with one point pulled far off it
y_outlier_raw = 0.60 * x + 2.6 + rng.normal(0, 0.15, len(x))
y_outlier_raw[3] += 9.74
y_outlier = match_stats(y_outlier_raw, target_mean, target_std)

# D: x is nearly constant except for one high-leverage point
rng2 = np.random.default_rng(1186)
x_leverage = np.full(len(x), 9.681)
x_leverage[:-1] += rng2.normal(0, 0.05, len(x) - 1)
x_leverage[-1] = 9.681 + 20.79
x_leverage = match_stats(x_leverage, x.mean(), x.std(ddof=1))

y_leverage_raw = target_mean + rng2.normal(0, 0.4, len(x))
y_leverage_raw[-1] = 11.896
y_leverage = match_stats(y_leverage_raw, target_mean, target_std)
```

Running that against the actual generated data confirms the match: all four datasets land at mean $x \approx 11.07$, mean $y \approx 9.69$, $\mathrm{std}(x) \approx 5.35$, $\mathrm{std}(y) \approx 3.70$, and $r \approx 0.73$, agreeing to two decimal places. Figure 9.1 plots all four, each with the identical best-fit line drawn on top.

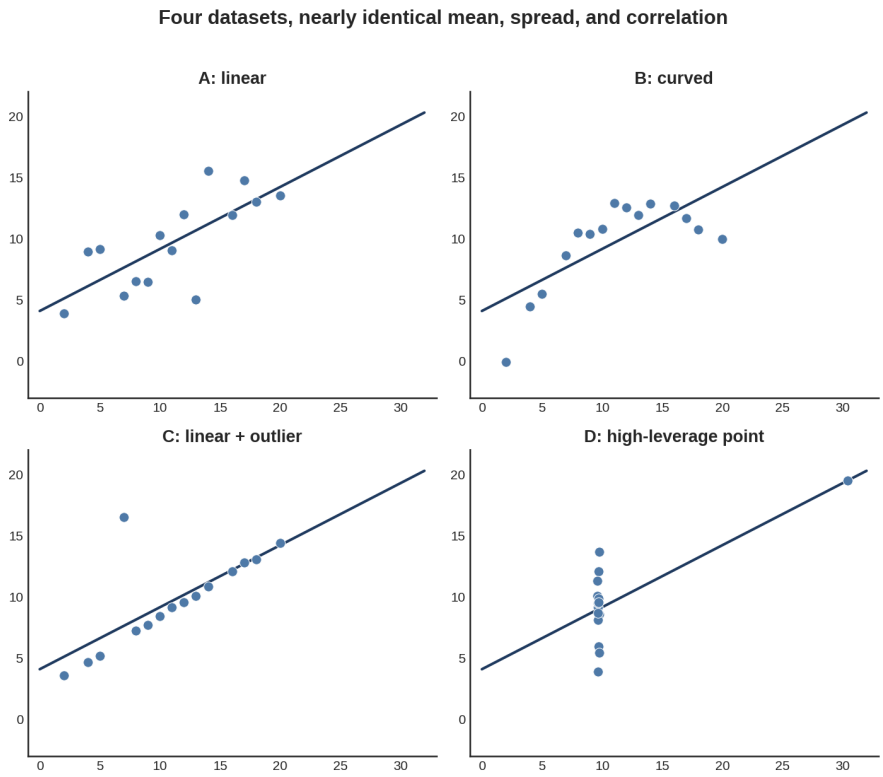


Figure 9.1: Four datasets built to share the same mean, spread, and correlation. A summary table would report the same four numbers for every one of them.

Panel A is the honest linear relationship the summary statistics describe. Panel B is a curve, not a line, that happens to produce the same correlation coefficient over this x-range. Panel C is the same line as A with a single point dragged far off it. Panel D is barely a relationship at all: fourteen points stacked at almost the same x-value, with one distant point single-handedly generating the entire correlation. A table of summary statistics reports the same four rows for all of them. Only the chart tells them apart.

9.4 Procedures

Procedure 9.1 (Profile a New DataFrame Fast). *When to use:* the first five minutes with any new dataset, before writing a single line of analysis.

Listing 9.2 A reusable fast-profiling function for any new dataframe

```
import pandas as pd

def quick_profile(df: pd.DataFrame) -> None:
    print(f"Shape: {df.shape[0]} rows x {df.shape[1]} columns")
    print(f"\nDtypes:\n{df.dtypes.value_counts()}")
    print(f"\nDuplicate rows: {df.duplicated().sum()}")

    missing = df.isnull().sum()
    missing = missing[missing > 0].sort_values(ascending=False)
    if len(missing):
        pct_missing = (missing / len(df) * 100).round(1)
        print(f"\nColumns with missing values:\n{pct_missing}")
    else:
        print("\nNo missing values.")

    print(f"\nNumeric summary:\n{df.describe().T}")

quick_profile(df)
```

Common mistake: running `.describe()` and stopping there. Summary statistics hide shape. Always follow up with a histogram, per Anscombe's quartet above.

Procedure 9.2 (Visualise the Missing-Value Pattern). *When to use:* immediately after `.info()` reveals nulls, to determine whether they are random or structural, e.g. a whole block of rows from one data source.

Listing 9.3 A missing-value heatmap: bright cells mark missing entries

```
import matplotlib.pyplot as plt
import seaborn as sns

fig, ax = plt.subplots(figsize=(12, 4))
sns.heatmap(
    df.isnull(),
    yticklabels=False,
    cbar=False,
    cmap='viridis',
    ax=ax,
)
ax.set_title('Missing Value Map (bright = missing)')
plt.tight_layout()
plt.show()
```

Procedure 9.3 (Plot a Distribution). *When to use:* for every numeric column, before deciding on transforms or outlier handling.

Listing 9.4 Histogram-plus-KDE and boxplot side by side for one numeric column

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

sns.histplot(
    df['fare'].dropna(), kde=True, ax=axes[0], color='#4E79A7'
)
axes[0].axvline(
    df['fare'].mean(), color='red', linestyle='--', label='mean'
)
axes[0].axvline(
    df['fare'].median(),
    color='orange',
    linestyle='--',
    label='median',
)
axes[0].legend()

sns.boxplot(x=df['fare'], ax=axes[1], color='#59A14F')
plt.tight_layout()
plt.show()
```

Common mistake: judging shape from a boxplot alone. Boxplots hide bimodality. A two-peaked distribution can look identical to a one-peaked one in a boxplot. Always pair it with a histogram or KDE.

Procedure 9.4 (Compute and Visualise a Correlation Matrix). *When to use:* once a dataset has more than a handful of numeric features and the full relationship structure needs to be seen at once.

Listing 9.5 Correlation heatmap plus a flag for near-redundant feature pairs

```
import numpy as np

corr = df[numeric_cols].corr(method='pearson')

fig, ax = plt.subplots(figsize=(8, 7))
mask = np.triu(np.ones_like(corr, dtype=bool))
sns.heatmap(
```

```
    corr,
    mask=mask,
    annot=True,
    fmt='.2f',
    cmap='coolwarm',
    center=0,
    square=True,
    linewidths=0.5,
    ax=ax,
)
plt.tight_layout()
plt.show()

# Upper triangle only: each pair once, no self-pairs
upper = corr.abs().where(
    np.triu(np.ones_like(corr, dtype=bool), k=1)
)
pairs = upper.stack()
print(pairs[pairs > 0.95].sort_values(ascending=False))
```

Common mistake: using Pearson correlation on a monotonic-but-non-linear relationship (e.g. $y = e^x$, where Pearson gives only about 0.69 for x from 0 to 10) and reading the score as a weak relationship. Switch to `method='spearman'` when a non-linear monotonic relationship is suspected, or look at the scatter plot. A U-shape, such as $y = x^2$ over both negative and positive x, scores near 0 on both. Only the scatter plot shows it.

Procedure 9.5 (Detect Outliers). *When to use:* after a distribution's shape has already been seen visually, when a repeatable, code-based flag is needed instead of eyeballing.

Listing 9.6 IQR-rule and z-score outlier flags as reusable functions

```
from scipy import stats

def iqr_outliers(series: pd.Series, k: float = 1.5) -> pd.Series:
    """Flag values outside [Q1 - k*IQR, Q3 + k*IQR]."""
    q1, q3 = series.quantile([0.25, 0.75])
    iqr = q3 - q1
    lower, upper = q1 - k * iqr, q3 + k * iqr
    return (series < lower) | (series > upper)

def zscore_outliers(
    series: pd.Series, threshold: float = 3.0
) -> pd.Series:
    """Flag values more than `threshold` std devs from the mean."""
    z = np.abs(stats.zscore(series.dropna()))
```

```
    flags = pd.Series(z > threshold, index=series.dropna().index)
    return flags.reindex(series.index, fill_value=False)

mask = iqr_outliers(df['fare'])
print(f"{mask.sum()} outliers ({mask.mean():.1%} of rows)")
```

Common mistake: deleting every flagged outlier automatically. An outlier is a data point to *investigate*, not always to remove: it might be a data entry error worth fixing or deleting, or a rare but valid case worth keeping so the model can learn from it.

Procedure 9.6 (Compare a Numeric Variable Across a Categorical Target). *When to use:* the single most useful bivariate check before modelling: does this feature separate the target classes?

Listing 9.7 Visual and statistical checks for whether a feature separates target classes

```
sns.kdeplot(
    data=df, x='fare', hue='survived', fill=True, common_norm=False
)

group_a = df[df['survived'] == 1]['fare'].dropna()
group_b = df[df['survived'] == 0]['fare'].dropna()
t_stat, p_value = stats.ttest_ind(
    group_a, group_b, equal_var=False
)
signal = (
    'likely informative' if p_value < 0.05 else 'weak/no signal'
)
print(f"p-value: {p_value:.2e}  ->  {signal}")
```

9.5 Worked Example

The tutorial notebook runs a complete, structured EDA on the Titanic passenger dataset (891 rows), following exactly the inspect-univariate-bivariate-multivariate-summarise structure above. The first inspection pass (Procedure 9.1 plus a missing-value heatmap, Listing 9.3) turns up three findings that shape everything downstream. `age` is missing for **19.9%** of passengers, `deck` for **77.2%**, and `embarked`/`embark_town` for 2 rows, few enough to drop safely. `age` needs imputation, and `deck` is close to unusable as a raw feature.

Target analysis comes next, because the target's own behaviour constrains every metric choice later in the book. Only **38.4%** of passengers survived. That's a real but moderate

imbalance, not the extreme kind that forces specialised techniques. But it's still enough that accuracy alone would be a misleading metric (Chapter 2's precision/recall framing).

Univariate analysis of the numeric features shows `fare` is heavily right-skewed (skewness ≈ 4.79: a small number of very expensive tickets stretch the distribution's tail), while `age` is close to symmetric (skewness ≈ 0.39). That skew is not a curiosity. It is the direct reason for log-transforming `fare` as a feature engineering step, an idea that resurfaces in Chapter 14.

Bivariate analysis against the target produces the chapter's clearest signal:

Listing 9.8 Survival rate by sex and by passenger class

```
df.groupby('sex')['survived'].mean()
df.groupby('pclass')['survived'].mean()
```

Run against the actual dataset, this returns a female survival rate of **74.2%** against a male survival rate of **18.9%**. That's a 55-point gap. It makes sex the strongest single predictor in the dataset, visibly reflecting the historical "women and children first" evacuation policy. Passenger class shows the same kind of gradient: **63.0%** survival in 1st class, **47.3%** in 2nd, and **24.2%** in 3rd. A Welch two-sample t-test comparing fares between survivors (mean £48.40) and non-survivors (mean £22.12) returns $p \approx 2.7 \times 10^{-11}$. That's an overwhelmingly significant difference, consistent with fare acting as a proxy for class and cabin location.

The multivariate correlation pass (Procedure 9.4) against `survived` quantifies what the bivariate charts already suggested: `pclass` correlates at -0.34 (higher class number, i.e. lower status, means lower survival), `fare` at $+0.26$, and `age` at only -0.08. That's a weak marginal effect on its own, one that the notebook's final multivariate step shows is misleading in isolation. Splitting age distributions by both class and survival status reveals a "children first" effect concentrated specifically in 1st and 2nd class, invisible in the marginal correlation and only visible once class is held fixed. That's exactly the kind of pattern multivariate analysis exists to surface.

The notebook closes not with a chart but with a written five-point summary. Target imbalance requires imbalance-aware metrics. Sex is the strongest predictor. Class interacts with sex, rather than acting independently. Fare needs a log transform. And age's 19.9% missingness should be handled with class-conditional median imputation, plus a `was_missing` indicator flag, rather than a single global fill value. That summary, not any individual chart, is the actual deliverable EDA hands to Chapter 14.

AWS SageMaker Equivalent

```
# SageMaker Data Wrangler generates a "Data Quality and
# Insights Report" (missing values, target leakage,
# duplicate rows, correlations, and outliers) without
# hand-written pandas.

# SageMaker Studio notebooks run the same pandas/matplotlib/
# seaborn workflow shown above, against data pulled directly
# from S3 or Athena:
import sagemaker
import pandas as pd

session = sagemaker.Session()
bucket = session.default_bucket()
# Requires s3fs installed
df = pd.read_csv(f"s3://{bucket}/raw/dataset.csv")
```

When to prefer SageMaker Data Wrangler (now part of SageMaker Canvas)**:** when a shareable, automatically-generated data quality report is needed for stakeholders, or when the dataset is too large to comfortably profile locally. For a single analyst doing hands-on exploration, notebook pandas and seaborn are faster to iterate with.

9.6 History and Further Reading

Thakur's early chapters give a practical, code-first approach to EDA that this chapter's procedures closely mirror (Thakur 2020). Burkov's second chapter covers the same territory from a more statistical angle, useful as a complementary framing of skewness, spread, and correlation (Burkov 2019, 15–25). Anscombe's quartet, four datasets sharing identical means, variances, and correlations yet visibly different in shape, is worth a five-minute read on its own. It's the single best reason for the "always visualise" rule in this chapter's Trade-offs section. The seaborn documentation gallery is also worth bookmarking directly. Each example chart answers a different EDA question, and the plot type decision guide in Appendix A cross-references it.

9.7 How This Connects

This chapter builds on Chapter 7 (data must be clean before EDA is meaningful) and Chapter 6 (pandas, matplotlib, and seaborn are the toolkit EDA runs on). It enables Chapter 14 (feature engineering, where EDA's findings become concrete transformations) and Chapter 11 (choosing which model families are worth trying, informed by what the data's shape and

relationships look like). It is often confused with feature engineering. But EDA is *observing*, while feature engineering is *transforming*. EDA's job is to inform the engineering decisions Chapter 14 makes. Not to make them.

9.8 Exercises

1. [**W**] List the five questions this chapter's EDA progression asks, in order, and explain in one sentence why "what surprises me?" is placed last rather than first.
2. [**W**] A dataset's `age` column has a mean of 29.7 and a median of 28.0. What does the gap between them suggest about the distribution's shape, and which single chart would confirm it?
3. [**A**] Using Listing 9.2 as a starting point, run it against a dataset of your choosing and report which columns, if any, have more than 10% missing values.
4. [**A**] Using Listing 9.8's grouping pattern, compute the Titanic survival rate broken down by `embarked` instead of `sex`. State whether the spread across categories looks large enough to be a useful feature.
5. [**A**] Reproduce Listing 9.5's "near-redundant pair" check on the Titanic numeric features (`survived`, `pclass`, `age`, `sibsp`, `parch`, `fare`) and confirm whether any pair exceeds the 0.95 threshold.
6. [**E**] The chapter found that `age`'s marginal correlation with `survived` was only -0.08, yet a clear "children first" effect appeared once the data was split by class. Propose one other pair of Titanic features where you'd expect a similar effect, hidden in the overall numbers until the data is split. Explain your reasoning before checking it.
7. [**E**] A colleague proposes running full EDA, including test-set rows, before splitting the data, arguing "we're just looking, not modelling yet." Using this chapter's data-snooping discussion, write two or three sentences explaining what could go wrong.
8. [**R**] Read about Anscombe's quartet (or its modern descendant, the "Datasaurus Dozen"). Pick one of the four Anscombe datasets. Describe, without plotting it, what summary statistics alone would have told you, and what they would have hidden.

9.9 Mental Model

EDA is journalism for datasets. The data scientist is investigating a story: gathering evidence, following leads, being surprised by what turns up, and then writing up findings in a way that helps someone else make a decision. That "someone else" is often future-you, deciding how to build the model.

10 The Data Science Lifecycle

The pre-modelling foundations are now in place: framing a business problem (Chapter 2), exploring and cleaning data (Chapters 7-9). Modelling starts next chapter. It's time for a change of pace. Knowing the individual skills is not the same as knowing how to run a project. The plan is more useful before the work than after it. Where do you start? When do you stop exploring and start modelling? When is a model "done"? What happens after it ships? Most data science failures are not algorithmic. They are process failures. A model built against the wrong version of the data, three weeks spent tuning before anyone checks whether the baseline was already good enough, a model that ships and silently degrades six months later with nobody watching. This chapter provides a map: the repeatable process that turns a business problem into a monitored, working system, and back into a better business problem.

10.1 Motivation

Every data science project moves through the same broad stages, whatever the use case. Understand the problem, get and understand the data, prepare the data, build a model, evaluate it, deploy it, monitor it, and use what was learned to do it better next time. If you read enough data science texts, you'll see this drawn as a clean left-to-right pipeline. It's not. Reality is a loop with paths going backwards at every stage. This chapter provides two things. A formal vocabulary for the stages, so data scientists can talk to each other through established frameworks like CRISP-DM. And a picture of how the process behaves in practice: non-linear, iterative, and full of opportunities for projects to quietly die.

10.2 Exposition

10.2.1 The Stages, Mapped to This Book

Stage	What happens	Where it lives in this book
1. Problem framing	Understand the business goal, translate it into a precise ML problem statement	Chapter 2
2. Data collection & understanding	Identify data sources, pull data, understand structure, quality, and limitations	Chapters 7-9
3. Data preparation	Clean, transform, and engineer features into a model-ready form	Chapters 7, 14
4. Modelling	Select an algorithm family, train candidate models, tune hyperparameters	Chapters 11-20
5. Evaluation	Measure model performance against the success metric defined in stage 1, not just accuracy, but the business metric	Chapters 13, 16
6. Deployment	Package the model and serve predictions, batch or real-time	Not in this book: Book 2, Chapter 23
7. Monitoring	Watch for data drift, model drift, and operational failures in production	Not in this book: Book 2, Chapter 25
8. Iteration	Feed what was learned, from evaluation and from production, back into a better problem statement	Back to stage 1, Chapter 2

This is a compressed version of two well-known industry frameworks. **CRISP-DM** (Cross-Industry Standard Process for Data Mining) runs through business understanding, data understanding, data preparation, modelling, evaluation, and deployment. Microsoft's **Team**

Data Science Process (TDSP) adds explicit emphasis on the operational stages (deployment, monitoring, project governance). The structure of this book series deliberately mirrors TDSP's operational emphasis, because a Solutions Architect is disproportionately valuable precisely at the stages CRISP-DM neglects.

10.2.2 Why the Textbook Diagram Lies

Every lifecycle diagram is drawn as a left-to-right pipeline with a feedback arrow tacked on to the end. That is not how the world works. In a real project, EDA (stage 2) routinely breaks the problem statement (stage 1): the label wanted doesn't exist, or the class balance makes the original success metric impossible to hit. Back to the drawing board before modelling has even begun. Modelling (stage 4) routinely sends a project back to data preparation (stage 3): a model's errors reveal a feature bug, or that an important signal isn't in the data. The two stages cycle dozens of times, rather than being "finished" in sequence. Evaluation (stage 5) routinely reveals that the success metric was wrong. A model hits 0.85 AUC. The team celebrates. Then the business owner explains they care about precision at a specific operating point nobody optimised for. That sends the project back to stage 1's definition of success. And monitoring (stage 7) is stage 1 for the next iteration: production drift is the start of the next project. Well-run teams treat monitoring alerts as new problem statements rather than tickets to patch.

Reality is a graph. Every stage has an edge back to every earlier stage. Good project management in data science is about recognising which stage you are in, not pretending you are close to some mythical end.

10.2.3 Why Projects Die at Each Stage

Stage	Common failure mode
Problem framing	No one asks "is ML even the right tool?" Months are spent on a problem a SQL query would have solved
Data understanding	The team never checks whether labelled data still reflects the current business process
Data preparation	Data leakage goes undetected until production, when the leaked feature no longer exists at prediction time
Modelling	The team optimises the model metric and never revisits whether it maps to the business metric agreed in stage 1

Stage	Common failure mode
Evaluation	The model is evaluated once, on one static test set, and never re-evaluated as new data arrives: silent staleness
Deployment	The model works in the notebook but nobody built the serving path, so it never reaches production: oddly, this is the most common failure in industry
Monitoring	No one owns the model after it's shipped; drift goes undetected for months
Iteration	"Done" is treated as permanent; no process triggers retraining or re-framing

Deployment is the stage most projects never reach. Industry surveys consistently find that most data science prototypes never make it to production. Which is why deployment, MLOps, and monitoring exist as a dedicated block in Book 2. The modelling techniques in Chapters 11-20 are necessary but not enough. A Solutions Architect's systems-thinking background is what's needed: "does it run in a notebook" and "is it a reliable production system" are not the same bar. That distinction is second nature to anyone who has built production systems.

10.3 Trade-offs and Limits

There is a trade-off between rigid process and iteration speed. Formal frameworks like CRISP-DM and TDSP are excellent for communicating with stakeholders and for audits. Followed literally as a waterfall, they slow projects down. Their practical use is as shared vocabulary and checklist, not as a gate that must be cleared before moving to the next stage.

> **i** Waterfall Wasn't Meant to Be Rigid Either
>
> Software engineering had this same argument decades earlier. Winston Royce's 1970 paper is usually credited as the origin of "waterfall," but Royce never used that word. His own paper calls the strict sequential version risky, and proposes feedback loops back to earlier phases as a required part of the process, not a sign something went wrong. The rigid, one-way reading came later, from people summarising the paper rather than reading it closely. The Agile Manifesto (2001) formalised the fix decades on: short iterative cycles instead of long sequential phases, built specifically to solve

the waterfall problem software teams had been living with since Royce's warning went unheeded. CRISP-DM and TDSP inherit the identical tension: a diagram drawn as a sequence, describing a process that was always meant to loop back.

There is a matching trade-off between depth of upfront framing and speed to first result. Spending three weeks perfecting a problem statement before touching data is its own failure mode. Nobody can know what's realistic until the data has been seen. The practical approach is a *lightweight* problem statement (Chapter 2's template, filled in quickly), followed by a fast pass through EDA, followed by a *revised* problem statement. Iterate the contract. Don't treat the draft as final.

Rabbit, Horse, or Elephant?

Robertson and Robertson's *Mastering the Requirements Process*, already the standard reference cited in Chapter 2, gives this trade-off a name (Robertson and Robertson 2012). Their Formality Guide sorts projects into Rabbit, Horse, and Elephant, with Horse as the default middle case. A Rabbit project is small, fast, and short-lived: it discovers requirements in small units, implements a small increment, and uses that working increment to get feedback from stakeholders, rather than writing a complete specification first. An Elephant project is the opposite: long duration, many distributed stakeholders, and a genuine need for a complete, formal specification before work starts. Data science projects are Rabbits, or should be treated as such whenever possible. The failure mode this section describes is treating one like an Elephant: writing the specification an Elephant needs for a project the data will reshape within a week.

10.4 Procedures

Procedure 10.1 (Write a Project Charter). *When to use:* at project kick-off, before any data work: a lightweight companion to Chapter 2's problem statement template, scoped to the whole project rather than the ML problem alone.

Listing 10.1 A project charter template, filled in before data work begins

```
## Project Charter

**Project name:** [Name]
**Sponsor / business owner:** [Who owns the outcome and
  signs off on "done"]
**Lifecycle stage owners:**
  - Data:        [who owns data access/quality]
```

```
  - Modelling:   [who builds and evaluates models]
  - Deployment:  [who owns the serving path]
  - Monitoring:  [who gets paged when it breaks -- must
    be a named person, not "the team"]

**Problem statement:** [link to the Chapter 2 ML problem
  statement]

**Timeline:**
  - Data understanding complete by: [date]
  - First baseline model by: [date]
  - Go/no-go decision by: [date]
  - Production deployment by: [date]

**Definition of done:** [Not "model built" -- "model
  deployed, monitored, and business metric moving"]

**Kill criteria:** [Under what conditions does this project
  get cancelled? e.g. "baseline model doesn't beat the
  existing rule by 10% relative"]
```

Common mistake: skipping "kill criteria." Projects without an explicit exit condition tend to survive purely on sunk cost, consuming months of effort on a problem that was never going to work.

Procedure 10.2 (Run a Pre-Mortem). *When to use:* right after the charter is written, before data work begins: a 30-minute exercise that surfaces risks while they are still cheap to fix.

Listing 10.2 A pre-mortem checklist, worked through before data work starts

```
## Pre-Mortem: Assume this project failed.
## It's 6 months from now. Why?

1. Data reasons it failed:
   - [ ] The label wasn't recorded reliably
   - [ ] Class imbalance was worse than expected
   - [ ] A key feature isn't available at prediction time
         (leakage discovered late)
   - [ ] Data access/pipeline broke and nobody noticed for weeks

2. Modelling reasons it failed:
   - [ ] The offline metric didn't translate to a business outcome
   - [ ] The model needed retraining more often than
         anyone budgeted for
```

```
3. Organisational reasons it failed:
   - [ ] Nobody owned deployment, so it stayed in a notebook
   - [ ] Nobody owned monitoring, so drift went undetected
   - [ ] The business owner changed and the new owner didn't buy in

4. For every checked box: what's the mitigation, and who owns it?
```

Common mistake: treating the pre-mortem as a formality and skipping straight to "no risks, let's build." The exercise only works if the team tries to find the failure mode before it happens.

Procedure 10.3 (Decide "Is This Even a Project?"). *When to use:* when a stakeholder brings an ask and it needs quick triage: full lifecycle project, or a five-minute answer?

Listing 10.3 A quick triage function for an incoming data science request

```python
def triage(ask: dict) -> str:
    """Quick triage for an incoming DS/ML request."""
    if ask.get("simple_rule_suffices"):
        return (
            "Not a project -- write the rule, ship it "
            "in a day (Chapter 2)"
        )
    if ask.get("one_off_analysis_only"):
        return (
            "Not a lifecycle project -- this is an "
            "analytics/EDA task (Chapter 9)"
        )
    if not ask.get("historical_labelled_data_exists"):
        return (
            "Not ready -- this is a data collection "
            "project first, an ML project second"
        )
    return "Full lifecycle project -- use the charter template"

ask = {
    "simple_rule_suffices": False,
    "one_off_analysis_only": False,
    "historical_labelled_data_exists": True,
}
print(triage(ask))
```

Procedure 10.4 (Structure a Data Science Project Repository). *When to use:* at the start of every project: a consistent layout makes handoffs and audits far easier.

Listing 10.4 A standard data science project repository layout

```
project_name/
|-- README.md      # Charter, kept up to date
|-- data/
|   |-- raw/       # Never modified -- read-only
|   |-- interim/   # Intermediate cleaning outputs
|   `-- processed/ # Model-ready datasets
|-- notebooks/
|   |-- 01_eda.ipynb
|   |-- 02_feature_engineering.ipynb
|   `-- 03_modelling.ipynb
|-- src/
|   |-- data_prep.py  # Reusable, tested code
|   |-- features.py
|   |-- train.py
|   `-- evaluate.py
|-- models/        # Serialised artefacts, versioned
|-- monitoring/
|   `-- drift_checks.py
|-- tests/
`-- requirements.txt
```

Common mistake: leaving all logic in notebooks. Notebooks are for exploration (stages 2-4); anything that needs to run in production (data prep, feature engineering, inference) belongs in tested `.py` modules that the notebooks *call*, not duplicate.

Procedure 10.5 (Run a Project Retrospective). *When to use:* at the end of every lifecycle iteration, whether the project shipped, was killed, or is looping back to stage 1.

Listing 10.5 A project retrospective template

```
## Retrospective: [Project Name] -- [Date]

**What we set out to do:** [Original problem statement]
**What actually happened:** [Outcome -- shipped / killed /
  iterating]

**What went well:**
- [...]

**What we'd do differently:**
- [...]
```

```
**Where did we loop back, and why?**
  - [Stage] -> [Earlier stage]: [reason]

**Is there a next iteration? If so, what's the revised
  problem statement?**
- [...]

**Action items for the team's process (not just this project):**
- [...]
```

Common mistake: only running retrospectives on failed projects. Successful projects have just as much to teach about which stage transitions worked smoothly and why. Capture that too.

10.5 Worked Example

The tutorial notebook carries a single churn-prediction problem, the same telco scenario from Chapter 2, through all eight lifecycle stages in miniature, in one notebook. It is worth telling because the result is a project that does *not* clear its own bar. That's the realistic outcome of a first pass, not a flaw in the exercise.

Stage 1 restates the abbreviated problem statement from Chapter 2: predict `churned_within_30d`, one row per customer at end of billing cycle. Success is defined as **precision $\geq$ 0.40 at recall $\geq$ 0.60** against the model metric, and beating the existing simple rule against the business metric. Stage 2 generates a synthetic population of 8,000 customers with a churn-generating process built from support tickets, payment failures, login recency, tenure, plan type, and product count. The resulting churn rate comes out to **7.0%**. That's realistically imbalanced, and low enough that accuracy alone would be almost meaningless as a metric, exactly the trap Chapter 2 warns against.

Stage 3 splits the data (80/20, stratified, `random_state=42`) before touching anything further: 6,400 training rows and 1,600 test rows, each preserving the 7% churn rate. It fits a `StandardScaler` on the training fold only, to avoid leaking test-set statistics into the features. Stage 4 then insists on two baselines before any real model is trained:

Listing 10.6 Two baselines established before any real model is trained

```
dummy = DummyClassifier(
    strategy='most_frequent', random_state=RANDOM_SEED
)
dummy.fit(X_train, y_train)

# Looser than Chapter 2's rule (> 2 tickets).
# X_test is unscaled: the scaler writes X_test_scaled.
rule_preds = (
    (X_test['num_support_tickets_90d'] >= 2)
    & (X_test['payment_failures_6m'] >= 1)
).astype(int)
```

On the actual generated data, the naive "always predict no churn" baseline scores 93% accuracy while catching **zero** churners (precision and recall both 0.00 on the positive class). That's a direct example of why accuracy is the wrong metric here. The hand-written rule does slightly better on recall but is still weak: 9% precision, 8% recall. Only then does Stage 4 train a logistic regression with class weighting, which reaches **AUC-ROC $\approx$ 0.638**, a real but modest improvement over guessing.

Stage 5 is where the lifecycle's honesty pays off. Sweeping thresholds for the point that clears recall ≥ 0.60 finds a threshold of $\approx$ **0.493**. But the precision achieved there is only $\approx$ **0.103**, well short of the 0.40 target set in Stage 1. At that threshold the model flags 661 of 1,600 test customers, more than six times as many as the simple rule's 100, for a precision barely above the rule's. **The model does not meet the business contract.** This is exactly the "loop-back" moment the chapter's Exposition describes in the abstract, made concrete. With only six raw features, the right move is not to declare victory on a good-looking AUC. It's to return to Stage 3 for better features (Chapter 14's feature engineering), or to Stage 1 to renegotiate the metric, not to proceed to deployment.

The notebook proceeds through Stages 6-8 anyway, explicitly labelled as showing *what those stages look like*, rather than a validated next step for this particular model. A deployment plan favours nightly batch scoring, since the intervention workflow takes hours, so real-time serving doesn't help. A monitoring plan watches feature drift, predicted-churn-rate drift, and realised precision on a rolling window. And a retrospective records the loop-back: Stage 5 to Stage 3 for better features, and potentially Stage 5 to Stage 1 if better features still don't clear the bar. The retrospective is a Python dictionary in the notebook, not a prose afterthought. That matters: writing the loop-back down in a structured form, easy to find again later, is what makes the next iteration happen instead of quietly dying. It echoes Procedure 10.1's kill criteria and named owners.

AWS SageMaker Equivalent

```
# SageMaker Projects gives a templated, governed version of
# the repo structure above, with CI/CD wiring for the
# deployment and monitoring stages built in.
import sagemaker

# Projects are typically created via the SageMaker Studio UI
# or Service Catalog, using a pre-built or custom MLOps
# template (build/train/deploy pipeline).
# Once created, each project provisions:
#   - A Git repo (CodeCommit, or GitHub/GitLab/Bitbucket
#     through a connection) mirroring the structure above
#   - CodePipeline stages for build -> train -> deploy
#   - A Model Registry entry per trained model version

# For a lighter-weight equivalent without full SageMaker
# Projects overhead, a SageMaker Pipeline captures the
# stages directly in code:
# from sagemaker.workflow.pipeline import Pipeline
# pipeline = Pipeline(
#     name="churn-lifecycle-pipeline",
#     # mirrors stages 3-6
#     steps=[
#         data_prep_step, train_step, eval_step, register_step
#     ],
# )
```

When to prefer SageMaker Projects: once a project has moved from "one person's notebook" to "a system with a defined deployment and monitoring owner", i.e., once it has cleared its own pre-mortem and is heading to production. Don't provision full MLOps scaffolding for stage-1/2 exploratory work.

10.6 History and Further Reading

The original CRISP-DM paper by Wirth and Hipp remains the most widely cited process framework in the field. It's worth reading in full, even though it predates the operational emphasis this chapter argues for. Microsoft's public documentation on the Team Data Science Process is the closest published equivalent to this chapter's own stage structure, particularly its explicit treatment of deployment and monitoring as first-class stages rather than afterthoughts. Thakur's opening chapter gives a practitioner's ground-level view of what a data science project looks like day to day, complementing this chapter's more structural framing (Thakur 2020). Burkov's first chapter, read alongside this one, gives the formal anatomy of a supervised learning project that this chapter's Stage 3-5 sequence builds directly on (Burkov 2019, 1–15).

10.7 How This Connects

This chapter builds on Chapter 2, which is where the lifecycle starts and which gets revisited every time the lifecycle loops back. It also builds on Chapter 9, whose EDA structure fills stage 2. It enables Book 2's chapters on experiment tracking, deployment, MLOps, and monitoring, pointing ahead to why each of those exists and where it fits before any of them are covered in depth. It is often confused with a project plan or Gantt chart. But the lifecycle is not a schedule. It is a project state machine, detailing what states a project can be in and how it moves between them. Two projects following the exact same lifecycle can take a week or a year, depending on how many loops back they need.

10.8 Exercises

1. [**W**] Name the eight lifecycle stages in order. For each, name one chapter (or block of chapters) in this book where that stage's skills live.
2. [**W**] Explain in two or three sentences why the textbook left-to-right pipeline diagram is misleading, using at least one concrete example of a backward loop from this chapter.
3. [**A**] Using Listing 10.3, construct an `ask` dictionary describing a hypothetical stakeholder request where `historical_labelled_data_exists` is `False`. Report what the function returns, and why that's the correct triage outcome.
4. [**A**] Using the logic of Listing 10.6 as a model, compute what accuracy the naive "predict majority class" baseline would score on a dataset with a 3% positive rate. Explain why that number is a poor argument for using the naive baseline in production.
5. [**A**] The tutorial's logistic regression achieved AUC-ROC of approximately 0.638 but failed to meet the precision/recall target from Stage 1. Using this chapter's Stage 5 discussion, name the two specific stages the project should loop back to. For each, give one concrete change that might close the gap.
6. [**E**] Rewrite the tutorial's Stage 6 deployment plan for a scenario where the business owner instead wants real-time scoring at the moment a customer contacts support, rather than nightly batch scoring. What changes about the serving pattern, the latency budget, and the rollback plan?
7. [**E**] A team tells you they "follow CRISP-DM," but their retrospectives show every project has shipped a model that was never revisited after deployment. Using Appendix A's Framework Comparison, explain which stage CRISP-DM under-specifies that likely explains this pattern. Explain what TDSP would have required instead.

8. [**R**] Read the original CRISP-DM paper (Wirth & Hipp, 2000) or the Microsoft TDSP documentation. Compare its treatment of "iteration" to this chapter's Stage 8. Does the source treat iteration as a distinct, named stage with its own deliverable, or as an implicit property of the whole process? What does that difference suggest about how seriously each framework takes the "loop back" problem?

10.9 Mental Model

The data science lifecycle is not a straight road walked once. It is a loop walked many times. The skill is knowing which stage you are in now versus which stage you *think* you are in. If a model's metrics look great but nobody agreed on what "great" means for the business, the project is still in stage 1. No matter how much modelling code has been written. If a model is in production but no one is watching it, the project is not done. It just hasn't discovered it's back at the beginning.

11 Supervised Learning: Overview of Approaches

Every supervised learning algorithm does the same thing. It takes examples where the answer is already known (this tumour was malignant, this customer churned, this house sold for $420k). It learns a mapping from inputs to outputs that generalises to examples it hasn't seen. Dozens of algorithm families claim to solve it, each with a different idea of what that mapping should look like. No one algorithm wins every time. This chapter provides an overview of these algorithms: what the major families assume, where each one excels, and the trade-off, bias versus variance, that explains why picking an algorithm is picking a set of assumptions about your data.

11.1 Motivation

A Solutions Architect doesn't need to have implemented every algorithm from scratch. But they do need enough working knowledge to answer a very practical question fast: given this dataset and this business constraint, what's the first thing worth trying? Reaching for the wrong family costs time. A KNN model that can't serve predictions inside a latency budget. A decision tree that overfits the moment it touches data. A linear model fit to a relationship that curves: wrong in the same way no matter how much data arrives. Every algorithm family encodes an **inductive bias**: an assumption about the shape of the relationship between inputs and outputs. There's no assumption-free algorithm. Choosing one is choosing what you're willing to believe about your data. Flexibility has its own cost. A neural network trained on 800 rows will usually lose to a logistic regression trained in a couple of minutes on the same data. Not because the network's assumptions are wrong. It barely has any. But 800 rows isn't enough data to pin down such an unconstrained model. Book 2 covers exactly why, once neural networks have been introduced.

11.2 Exposition

11.2.1 The Bias-Variance Trade-Off

This is probably the most important idea in supervised learning. Everything else in this chapter is downstream of it. Every model's error comes from two sources.

Bias is how wrong the model's assumptions are. Systematically. No matter how much data it gets. A linear model fit to a curved relationship stays wrong forever. More data doesn't fix a mismatched hypothesis class. It just makes the wrongness more precisely measured.

Variance is how much the model's predictions change when trained on a different sample from the same population. A model that memorises its training data exactly will produce a completely different fit if a handful of training examples change. That instability is variance.

$$\text{Expected Test Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

The irreducible error is noise in the data. No model removes it. The trade-off is between the other two. A simple model (a straight line) has high bias and low variance. It's wrong in the same way every time, but it's stable. A flexible model (an unpruned decision tree) has low bias and high variance. It can better represent the true relationship. But it also happily fits noise, and small changes in training data produce a very different fit. Model complexity is a dial. The goal is finding where on that dial test error is minimised. That point has to be found empirically, because it depends on the specific dataset. Chapter 13 (Cross-Validation & Model Selection) is the tool that finds it.

11.2.2 Linear Models

What they assume: the relationship between inputs and output is a straight line (regression) or that a hyperplane can separate the classes (classification).

> **i** What Is a Hyperplane?
>
> A hyperplane is the general-dimensional version of a line. In two dimensions it's an ordinary line. In three, an ordinary flat plane. In n dimensions, it's whatever flat, one-dimension-lower surface generalises that idea. It's defined by the same equation either way: $\mathbf{w} \cdot \mathbf{x} + b = 0$, using the dot product from Chapter 3. Every point on one side of that surface gives $\mathbf{w} \cdot \mathbf{x} + b > 0$. Every point on the other side gives a negative value. A linear classifier's whole job is choosing $\mathbf{w}$ and b so that surface falls between the two classes as cleanly as possible. A dataset with 30 features doesn't live

somewhere a hyperplane can be drawn and looked at. But the equation that defines one doesn't care how many dimensions it's operating in.

Logistic Regression is a classifier despite the name. It models $P(y = 1 \mid x)$ with a sigmoid applied to a linear combination of features, and its outputs are probabilities, not just labels, and often reasonably well calibrated ones.

What Does a Sigmoid Curve Look Like?

The sigmoid function is $\sigma(z) = \frac{1}{1+e^{-z}}$, and its job is squashing any real number into the open interval $(0, 1)$, so it can be read as a probability. Large positive z pushes $\sigma(z)$ towards 1; large negative z pushes it towards 0; $z = 0$ gives exactly 0.5. Plotted, it's a smooth S-shaped curve (Figure 11.1), steep near the middle and flat at both ends. That flatness matters: once the model is already confident, a further change to its linear combination of features barely moves the predicted probability, but near $z = 0$, a small change swings the prediction hard in one direction or the other.

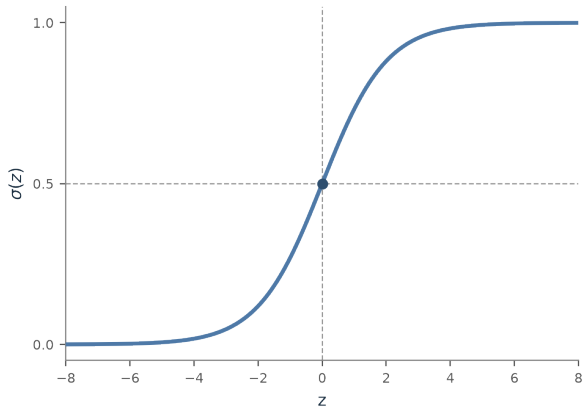


Figure 11.1: The sigmoid function: a smooth S-curve, steepest at z = 0, flattening as z moves towards either extreme.

Linear Regression fits a line by minimising squared error. Both are fast, interpretable (the coefficients are the explanation), and a recommended first step. Train the linear baseline before anything fancier. If a complex model can't beat your baseline, the problem is the features or the data, not the missing complexity. Weaknesses: no nonlinear relationships without manual feature engineering, sensitivity to outliers (linear regression), and unstable coefficients when features are strongly correlated with each other (multicollinearity), which regularisation only partly fixes.

> **What Is Regularisation?**
>
> Regularisation adds a penalty for complexity to what a model is trying to minimise, on top of prediction error. Instead of fitting the training data as closely as possible, the model is pushed towards smaller, simpler coefficients, which tends to generalise better and lean less heavily on any single feature. For linear regression, that means adding a cost for large weights to the loss function, so a handful of correlated features can't each grab an oversized, unstable share of the prediction the way they could unregularised. It doesn't remove the correlation. It just limits the damage. Chapter 18 covers regularisation properly, including its two standard forms, Ridge and Lasso, and why they behave differently.

11.2.3 Support Vector Machines

What they assume: among every hyperplane that separates the classes, the one with the widest margin to the nearest points of each class generalises best.

A **Support Vector Machine (SVM)** starts from the same hyperplane idea as Logistic Regression, but picks it differently. Instead of any separating hyperplane, it finds the one maximising the distance to the closest point of either class, the margin. Only those closest points, the support vectors, determine where the boundary sits. Every other point could move, or vanish, without changing the decision boundary at all. Real data is rarely perfectly separable, so a slack parameter C trades margin width against tolerating some misclassified or margin-violating points. Small C gives a wide margin that tolerates more violations: higher bias, lower variance. Large C gives a narrow margin that tolerates few: lower bias, higher variance. That's the same trade-off this chapter's first section named.

> **What Is the Kernel Trick?**
>
> A kernel function computes the dot product two points would have *after* being mapped into some higher-dimensional space, without ever performing that mapping explicitly. That matters because a boundary that's a curve in the original space can be a hyperplane in the right higher-dimensional one. SVM training only ever needs dot products between points, never their raw transformed coordinates. The RBF kernel, $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$, corresponds to an infinite-dimensional mapping and is the default choice when there's no reason to assume a specific functional form. This is exactly the trick that lets an SVM draw a curved boundary around data no straight line can separate, the concentric-circles case later in this chapter's Worked Example, where Logistic Regression collapses to 0.289.

SVMs handle high-dimensional data well, and remain effective even when the number of features exceeds the number of rows. But training scales poorly past roughly tens of

thousands of rows. And, like every distance-based method in this chapter, they need scaled features to behave sensibly.

11.2.4 Tree-Based Models

What they assume: the relationship is well-approximated by a sequence of if/else splits on individual features.

A **Decision Tree** recursively splits training data on whichever feature best separates classes, or reduces regression error, at each node. Each leaf is a prediction. Trees are naturally interpretable (print one and read it like a flowchart) and need no feature scaling. (scikit-learn's trees still need categories encoded as numbers.) Their weakness is the high-variance failure mode above: an unconstrained tree memorises its training set, and a small change in the data produces a structurally different tree. Ensembling many trees, covered in Chapter 12, is the standard fix and the reason trees remain the dominant building block for tabular data despite being weak individually.

11.2.5 Distance-Based Models

What they assume: similar inputs produce similar outputs: nearby points in feature space share a label.

k-Nearest Neighbours (KNN) classifies a new point by finding its k closest training examples, Euclidean distance by default, and taking a majority vote. For regression, it averages their targets. KNN makes no assumption about the shape of the boundary. It can represent arbitrarily complex relationships. But it pays for that flexibility at prediction time: every prediction requires computing distance to every training point, an $O(n)$ cost per prediction that gets expensive fast. It also degrades in high dimensions: the curse of dimensionality. As dimensionality grows, all points become roughly equidistant, and "nearest" loses meaning. And it requires feature scaling. Distance is meaningless when one feature ranges 0–1 and another 0–1,000,000.

11.2.6 Probabilistic Models

What they assume: features are conditionally independent given the class label, the "naive" in Naive Bayes.

That assumption is almost never true, and the algorithm works well anyway. **Naive Bayes** applies Bayes' theorem to compute the posterior probability of each class. The independence assumption reduces this to multiplying per-feature likelihoods, which is extremely fast to train and predict. It's a strong, cheap baseline for text classification (it works well on bag-of-words features, even though words are far from independent) and for any problem

with very little training data, since it has few parameters to estimate. It underestimates feature correlations and can't represent interactions, so it's rarely competitive as a final model. But as a five-minute sanity check, it's hard to beat.

11.2.7 The No Free Lunch Theorem

There's a formal result behind the intuition that "it depends." Averaged over *all possible problems*, no algorithm outperforms any other, including random guessing. What this rules out is a universal favourite. What it does not rule out, and what makes data science a skill rather than a lookup table, is that specific algorithms reliably outperform others on the specific *kinds* of problems that show up in practice. On tabular data, gradient-boosted trees (Chapter 12) win most of the time. On unstructured data, deep learning wins. Neither fact contradicts No Free Lunch, because "tabular data" and "image data" are not uniform samples over all possible problems. They have structure, and different algorithms are built to exploit different kinds of structure.

11.3 Trade-offs and Limits

Algorithm	Bias	Variance	Interpretable	Nonlinear	Speed
Logistic Regression	High	Low	Yes	No	Very fast
SVM (RBF kernel)	Low–Med	Low–Med	No	Yes	Slow to train at scale
Decision Tree (deep)	Low	High	Yes (if shallow)	Yes	Fast
KNN	Low	High	No	Yes	Slow at predict time
Naive Bayes	High	Low	Partial	No	Very fast
Random Forest	Low–Med	Low–Med	No	Yes	Medium

No row in this table is dominant. That's the No Free Lunch theorem showing up as a design constraint. The practical resolution is empirical: try the fast, interpretable baseline first, and let cross-validated performance decide whether the added complexity of a more flexible family gives you something useful.

11.4 Procedures

Procedure 11.1 (Train and Evaluate Any sklearn Classifier: the Universal Pattern). *When to use:* every classification task, regardless of algorithm: the API is identical across all five families above.

Listing 11.1 The universal sklearn fit/predict/evaluate pattern, always wrapped in a Pipeline

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# swap for any classifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=RANDOM_SEED, stratify=y
)

clf = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print(f'Accuracy: {accuracy_score(y_test, y_pred):.3f}')
print(classification_report(y_test, y_pred))
```

Why the Pipeline matters: it guarantees the scaler is fitted only on training data and applied, not refitted, on test data. Fitting the scaler on the full dataset before splitting is a leakage bug, the classification analogue of the leakage failure mode from Chapter 2.

Procedure 11.2 (Compare Multiple Classifiers Under Identical Conditions). *When to use:* deciding which algorithm family is worth investing tuning time in, before committing to one.

Listing 11.2 Benchmarking several classifiers with identical preprocessing and cross-validation

```
from sklearn.model_selection import cross_val_score

classifiers = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(
        random_state=RANDOM_SEED
    ),
    'K-Nearest Neighbours': KNeighborsClassifier(n_neighbors=5),
    'Naive Bayes': GaussianNB(),
    'Random Forest': RandomForestClassifier(
        random_state=RANDOM_SEED, n_estimators=100
    ),
    'SVM (RBF)': SVC(random_state=RANDOM_SEED),
}

for name, clf in classifiers.items():
    pipe = Pipeline([('scaler', StandardScaler()), ('clf', clf)])
    scores = cross_val_score(
        pipe, X_train, y_train, cv=5, scoring='accuracy'
    )
    print(f'{name:<25} {scores.mean():.3f} ± {scores.std():.3f}')
```

Common mistake: comparing algorithms using a single train/test split. Different splits can reorder the ranking entirely on a small dataset. Chapter 13 quantifies exactly how much.

Procedure 11.3 (Get and Threshold Prediction Probabilities). *When to use:* whenever a confidence score matters more than a hard label: risk scoring, ranking, or a business-specific decision threshold (the precision/recall trade-off from Chapter 2).

Listing 11.3 Extracting probabilities and applying a custom decision threshold

```
proba = clf.predict_proba(X_test)   # shape (n_samples, n_classes)
positive_class_proba = proba[:, 1]

threshold = 0.3   # lower than 0.5 → higher recall, lower precision
y_pred_custom = (positive_class_proba >= threshold).astype(int)
```

Not every estimator supports `predict_proba` (`LinearSVC` is a common exception). Use `decision_function()` instead, or switch to `SVC(probability=True)`.

Procedure 11.4 (Diagnose Bias vs. Variance from the Train/Test Gap). *When to use:* a model's cross-validated score is disappointing and the next step, more data, more regularisation, or a more flexible model, depends on which failure mode is present.

Listing 11.4 Comparing train and test accuracy to diagnose overfitting vs. underfitting

```
train_acc = accuracy_score(y_train, clf.predict(X_train))
test_acc  = accuracy_score(y_test, clf.predict(X_test))
gap = train_acc - test_acc
print(
    f'Train: {train_acc:.3f}  Test: {test_acc:.3f}  Gap: {gap:.3f}'
)
```

Rule of thumb: train high, test low, large gap → high variance (overfitting): simplify the model or get more data. Train and test both low → high bias (underfitting): use a more flexible model or better features. Wide spread across CV folds → high variance regardless of the train/test gap.

11.5 Worked Example

The tutorial notebook runs six classifiers (Logistic Regression, SVM, Decision Tree, KNN, Naive Bayes, Random Forest) on the sklearn breast cancer dataset. It has 569 tumour samples and 30 numeric features, labelled malignant (class 0) or benign (class 1), split 62.7% benign / 37.3% malignant. A stratified 80/20 split gives 455 training and 114 test samples.

Benchmarking all six with Procedure 11.2 (5-fold CV on the training set) plus a held-out test score and timing gives the results below. Timings are from this machine and vary by run and hardware. The CV and test scores are deterministic given the fixed seed.

Algorithm	Test Acc.	CV Mean ± Std	Train Time	Predict Time
Logistic Regression	0.982	0.980 ± 0.013	16.0 ms	1.1 ms
SVM (RBF)	0.982	0.971 ± 0.018	6.5 ms	1.5 ms
K-Nearest Neighbours	0.956	0.967 ± 0.021	1.5 ms	35.4 ms
Random Forest	0.956	0.954 ± 0.023	95.7 ms	5.8 ms
Naive Bayes	0.930	0.932 ± 0.004	1.7 ms	0.6 ms
Decision Tree	0.912	0.910 ± 0.019	7.8 ms	0.9 ms

Logistic Regression and SVM tie for the top test accuracy. That's unsurprising: 30 well-engineered clinical features separating two tumour classes is close to the linear-model sweet spot. The RBF kernel's added flexibility here doesn't help beyond what a straight hyperplane was already finding. Its slightly lower CV mean (0.971 vs. 0.980) hints at a marginally less stable fit for capacity the data didn't need. Two details are worth more attention than the ranking. First, KNN's prediction time (35.4 ms) dwarfs every other model's: more than six times slower than the next-slowest, Random Forest (5.8 ms), and over sixty times slower than the fastest, Naive Bayes (0.6 ms). That's the $O(n)$ cost from the Exposition section, visible directly in milliseconds rather than asserted. Second, Naive Bayes has by far the lowest CV standard deviation (± 0.004 against ± 0.013–0.023 for everything else). The independence assumption gives it less to overfit, trading a slightly lower mean for a much more stable estimate: exactly the low-variance side of the bias-variance table.

Random Forest's full classification report on the held-out test set (with all 30 features) reads: malignant precision 0.95, recall 0.93, f1 0.94 (support 42); benign precision 0.96, recall 0.97, f1 0.97 (support 72); overall accuracy 0.96. The confusion matrix is `[[39, 3], [2, 70]]`: 3 malignant tumours misclassified as benign, 2 benign misclassified as malignant. The costlier error clinically, a missed malignancy, is the more common one here: 3 against 2. That's exactly why Chapter 2's point about business-owned thresholds matters, even on a textbook dataset.

Restricting every classifier to two features (`mean radius`, `mean texture`) so the decision boundary can be plotted in 2D drops accuracy across the board. Logistic Regression, KNN, and Naive Bayes all land at 0.877, Random Forest at 0.860, and Decision Tree falls hardest, to 0.789. The interesting result isn't the accuracy drop: that's expected, since 28 informative features were discarded. It's that three structurally different algorithms converge on the *same* accuracy with only two features. The tree, the algorithm best able to exploit many features via deep splits, suffers the most from losing them, since it has nothing left to split on beyond two axes.

Finally, a direct empirical example of No Free Lunch: on a synthetic dataset of two linearly-separable blob clusters, both Logistic Regression and Random Forest score a perfect 1.000. On a synthetic dataset of two concentric circles, no linear boundary can separate them, Random Forest still scores 1.000. Logistic Regression collapses to **0.289**. That's worse than the ~0.5 a coin flip would give on balanced classes. With features standardised around the same origin the two circles share, the linear decision boundary lands on the *wrong* side of the split, actively anti-correlating with the true label rather than merely failing to find one. Two algorithms, two datasets, and the ranking flips completely. The No Free Lunch theorem is not an abstraction here. It's a number that goes from 1.000 to 0.289 depending on the shape of the data.

11.6 History and Further Reading

Burkov's Chapter 3 gives a concise, precise pass through the main algorithm families this chapter covers. Chapter 4 formalises "the anatomy of a learning algorithm," the representation/loss/optimisation breakdown this chapter's Motivation section leans on (Burkov 2019). Murphy's Introduction, Chapters 9–10, gives the full probabilistic treatment of linear and logistic regression for readers who want the derivations behind the sigmoid and the squared-error loss (Murphy 2022). Géron's *Hands-On Machine Learning* devotes separate chapters to classification (Ch. 3), support vector machines (Ch. 5), and decision trees (Ch. 6), each with worked Scikit-Learn code for the families this chapter surveys (Géron 2019). The supervised-learning No Free Lunch theorem is due to Wolpert (1996). Wolpert and Macready (1997) proved the companion result for optimisation. It's worth reading in the original for how narrow its actual claim is: far narrower than the popular one-line summary suggests.

11.7 How This Connects

This chapter builds on Chapter 3 (the probability and linear algebra underneath every algorithm here), Chapter 6 (the Python/sklearn ecosystem all the code above assumes), and Chapter 10 (the lifecycle stage this sits inside). It enables Chapter 12, where these same algorithms (trees especially) become base learners inside ensembles, and Chapter 16, which needs trained models before it can evaluate them. It's often confused with the supervised/unsupervised distinction: everything in this chapter learns from *labelled* examples. Chapter 19 covers what happens when there's no label to learn from at all.

11.8 Exercises

1. **[W]** For each of the following, name the algorithm family (Linear, Support Vector Machine, Tree-Based, Distance-Based, Probabilistic) and state its core assumption in one sentence: (a) spam detection using word-count features; (b) predicting house price from square footage and location; (c) classifying flowers from petal measurements using their five closest labelled neighbours.
2. **[W]** Explain, in your own words, why increasing model flexibility (e.g., letting a decision tree grow deeper) tends to reduce bias while increasing variance. Use the bias-variance equation from this chapter.
3. **[A]** Using the benchmark table in the Worked Example, identify every algorithm whose *test* accuracy exceeds its *CV mean* accuracy. Explain what that ordering does, or doesn't, tell you about overfitting for those models.

4. **[A]** The Worked Example shows Decision Tree accuracy falling from 0.912 (30 features) to 0.789 (2 features), the largest drop of any algorithm. Using this chapter's description of how trees split, explain why restricting to two features hurts a tree more than it hurts Naive Bayes (0.930 → 0.877).
5. **[E]** A client has a 40,000-row tabular dataset with strongly nonlinear feature interactions and needs predictions to run inside a 5ms latency budget per request. Using Listing 11.1 and this chapter's Trade-offs table, name the algorithm you would rule out first, and explain why using the prediction-time numbers from the Worked Example.
6. **[E]** The No Free Lunch theorem is often summarised as "no algorithm is better than any other." Using the blobs-vs-circles result (1.000 vs. 0.289 for Logistic Regression), explain why that popular summary is misleading for a working data scientist, and state the narrower claim the theorem makes.
7. **[R]** Read Burkov's Chapter 4 on the anatomy of a learning algorithm. Apply its three-part framework (hypothesis representation, loss function, optimisation procedure) to Naive Bayes, and explain what makes its optimisation step so much faster than logistic regression's.

11.9 Mental Model

Every supervised learning algorithm is a machine for turning labelled examples into a decision rule. The algorithms differ only in what shape of rule they're willing to consider: a line, a maximum-margin boundary (straight or kernel-bent), a set of rectangular cuts, a neighbourhood vote, a product of per-feature likelihoods. Before touching code, ask what shape the real boundary between classes (or the true curve through the regression data) is likely to take. The answer to that question picks the first algorithm to try faster than any benchmark table can.

12 Ensemble Methods

A tree standing alone is easily blown down in a storm, where a forest continues to stand. A single decision tree is weak in the same way. Train it on slightly different data, and the result is a completely different tree. Train it deep enough to capture real complexity, and it overfits. Keep it shallow enough to generalise, and it underfits. Yet tree-based algorithms, combined into ensembles, produce the most accurate models for tabular data that exist. This chapter is about why combining many mediocre models consistently beats any single good one, and how that principle plays out differently in bagging, boosting, and stacking.

12.1 Motivation

The underlying intuition is old: wisdom of crowds. In 1906, statistician Francis Galton looked at about 800 guesses of an ox's weight from a county fair contest. The median guess landed within 1% of the true weight, closer than most of the individual guesses. Individual errors cancelled out in the aggregate. Ensemble methods apply the same principle to machine learning models. But it only works if the individual models make *different* errors. If every model in the ensemble fails on the same examples, averaging doesn't help. The diversity of errors is what produces the advantage. There are two main strategies for creating that diversity: bagging and boosting.

12.2 Exposition

12.2.1 Bagging — Reducing Variance

Bagging stands for Bootstrap AGGregating. Sample n examples from the training set with replacement. Each example goes back into the pool before the next draw, so some appear multiple times and some not at all. Train a model on this bootstrap sample, repeat B times, then average the regression outputs (or majority-vote the classification outputs). Each model sees a slightly different dataset and makes slightly different errors. Averaging cancels the errors and amplifies the signal. If B models each have variance σ^2 and their errors are uncorrelated, the average has variance σ^2/B: more models, lower variance, with no increase in bias.

12.2.2 Random Forests — Bagging Plus Feature Randomness

Random Forests add a second source of diversity on top of bagging: at each split in each tree, only a random subset of the features is considered: roughly $\sqrt{p}$ of p for classification. (scikit-learn's `RandomForestRegressor` defaults to all features, which makes it plain bagging unless `max_features` is set.) This sounds like a bad idea. Why limit the information available to each split? Because without feature randomness, every tree in a bagged ensemble tends to pick the same dominant feature at the root, producing correlated trees whose errors don't cancel. Feature randomness decorrelates the trees. Decorrelated trees produce more diverse errors. More diverse errors cancel better when averaged. That's why Random Forests are usually more accurate than plain bagging. Random Forests are especially strong on tabular data with mixed feature types, when feature importance scores are wanted, and when there's little appetite for hyperparameter tuning.

12.2.3 Boosting — Reducing Bias

Boosting trains models sequentially rather than in parallel, with each new model focused on the previous ensemble's mistakes. The original recipe, AdaBoost: train a weak learner (typically a shallow "stump"), increase the weight of misclassified examples, train the next learner on the reweighted data so it focuses on the hard cases, repeat, and combine as a weighted sum. Boosting mainly reduces bias, systematic error. Gradient boosting's shrinkage and subsampling also keep its variance in check. It's especially effective when a model is wrong in a *correctable* way.

12.2.4 Gradient Boosting — Boosting as Gradient Descent

Modern gradient boosting (Friedman, 2001) reframes boosting as gradient descent in function space. Ordinary gradient descent (Chapter 4) nudges a set of numbers, the weights, in the direction that most reduces the loss. Gradient boosting nudges a whole function instead: each new tree is the "step," added to the ensemble because it points the current prediction function in the direction that reduces error the most. Instead of reweighting examples, each new tree is trained to predict the residual errors of the current ensemble (for squared error; for other losses, the negative gradient of the loss): fit tree_1 to predict y, compute $\text{residuals}_1 = y - \text{tree}_1(X)$, fit tree_2 to predict residuals_1, and continue. Each tree corrects what the ensemble currently gets wrong, and the training error falls with each round. The learning rate (shrinkage) controls how aggressively each tree corrects: small learning rates need more trees but generalise better.

12.2.5 XGBoost and LightGBM

XGBoost (2016, Chen & Guestrin) is an engineered gradient boosting implementation adding built-in L1/L2 regularisation on leaf weights, pruning (grows trees then prunes back, rather than greedy early stopping), native handling of missing values, row/column subsampling, and parallelised split-finding. Its paper reports that 17 of the 29 winning Kaggle solutions published in 2015 used it.

LightGBM (2017, Microsoft) improves training speed in three ways. It grows trees leaf-wise instead of level-wise, finding whichever split reduces loss most, whatever the resulting depth. It also uses Gradient-based One-Side Sampling (keeping high-gradient examples, sampling low-gradient ones), and Exclusive Feature Bundling (merging mutually exclusive sparse features). Its original paper reports training up to 20x faster than earlier gradient boosting implementations on large datasets.

The key insight for a Solutions Architect: on tabular data (the vast majority of enterprise ML), gradient boosting (XGBoost or LightGBM) tends to beat neural networks on accuracy. It also trains faster, is easier to interpret via SHAP, and costs less to run. Benchmark against LightGBM before reaching for a deep learning solution on structured data.

> **What Is SHAP?**
>
> SHAP (SHapley Additive exPlanations) breaks a single prediction down into each feature's contribution, in a way that's mathematically guaranteed to add up to the gap between that prediction and the model's average prediction, for any model type. For a rejected loan application, SHAP can say exactly how many percentage points "low credit score" or "short employment history" each contributed, not just that the model used those features somewhere. What isn't guaranteed everywhere is exactness of the individual numbers. Computing the true Shapley value means checking every possible subset of features, exponential in general. Tree-based ensembles are the exception: `TreeExplainer` computes exact values in polynomial time, by exploiting the tree structure directly. That's a real part of why they're easier to interpret in practice than a neural network of comparable accuracy, whose SHAP values come from a slower approximation instead. Book 2, Chapter 21 covers SHAP properly, including how to compute and read it.

12.2.6 Stacking — Learning How to Combine

Stacking (Wolpert, 1992) trains a meta-learner to decide how to weight the base models' predictions, rather than averaging or majority-voting them. Train K base models using cross-validation. Use their out-of-fold predictions as features for a meta-learner (typically logistic or ridge regression). Train the meta-learner on those features against the true labels. For new data, generate base predictions and pass them to the meta-learner. Stacking is most

valuable in competition settings. The complexity it adds is often not worth it in production, and, as this chapter's Worked Example shows directly, it is not automatically better than the best individual model.

12.3 Trade-offs and Limits

Method	Reduces	Risk	Best for
Bagging	Variance	None inherent	High-variance base learners (deep trees)
Random Forest	Variance	Slow on very wide data	Robust tabular baseline
Gradient Boosting	Bias + Variance	Overfitting if trees too deep / learning rate too high	Most tabular tasks
Stacking	Both, in principle	Complexity, leakage if done wrong, no guarantee of improvement	Squeezing the last per cent in competitions

Random Forests and boosted trees both work well on tabular data. But they are not interchangeable defaults. Random Forests are more forgiving of minimal tuning. Gradient boosting variants need more care, learning rate, tree depth, number of estimators, to reach their best accuracy. That same sensitivity is what lets them beat Random Forests once tuned.

12.4 Procedures

Procedure 12.1 (Tune `n_estimators` and `max_depth`). *When to use:* after choosing an ensemble method, to improve its accuracy.

Listing 12.1 Sweeping n_estimators with cross-validation

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score

for n in [50, 100, 200, 300]:
    scores = cross_val_score(
        RandomForestRegressor(
            n_estimators=n, random_state=42, n_jobs=-1
        ),
        X_train,
        y_train,
        cv=5,
        scoring='neg_root_mean_squared_error',
    )
    rmse = -scores.mean()
    print(f'n={n:>3}  RMSE={rmse:.0f} ± {scores.std():.0f}')
```

Guidance: for Random Forest, `n_estimators=100` is a safe default, and going beyond 300 rarely helps. For XGBoost/LightGBM, use more trees (500-1000) with a smaller learning rate (0.05). `max_depth=4` is a reasonable starting point for gradient boosting. Random Forest typically uses unlimited depth (`None`).

Procedure 12.2 (Plot Feature Importance). *When to use:* to explain which features drive a model's predictions.

Listing 12.2 Impurity-based and permutation-based feature importance

```
# Impurity-based, all sklearn ensembles
importances = model.feature_importances_

from sklearn.inspection import permutation_importance
result = permutation_importance(
    model, X_test, y_test, n_repeats=10, random_state=42
)
```

Common mistake: trusting impurity-based `.feature_importances_` alone. It's biased towards high-cardinality features. Permutation importance (recomputing performance with a feature shuffled) is slower but doesn't favour high-cardinality features, and should be the tiebreaker whenever the two disagree. It can still mislead when features are strongly correlated with each other.

Procedure 12.3 (Choose Between Random Forest and XGBoost/LightGBM). *When to use:* starting a new tabular modelling task.

Listing 12.3 Decision path for a first ensemble choice

```
# No time to tune?
#   -> Random Forest (n_estimators=100, defaults work well)
# Training speed critical on a large dataset? -> LightGBM
# Missing values present?
#   -> XGBoost or LightGBM (handle natively)
# Need native categorical support?
#   -> LightGBM (categorical_feature=...) or CatBoost
```

Procedure 12.4 (Build a Simple Stacking Ensemble). *When to use:* squeezing the last 0.5-1% accuracy out of a set of already-good models, not as a default choice.

Listing 12.4 Stacking with out-of-fold predictions to avoid leakage

```
from sklearn.model_selection import cross_val_predict, KFold
from sklearn.linear_model import Ridge

kf = KFold(n_splits=5, shuffle=True, random_state=42)
base_models = {
    'rf': RandomForestRegressor(
        n_estimators=100, random_state=42, n_jobs=-1
    ),
    'lgb': lgb.LGBMRegressor(
        n_estimators=100, learning_rate=0.1, random_state=42
    ),
}

oof, test_preds = {}, {}
for name, model in base_models.items():
    oof[name] = cross_val_predict(model, X_train, y_train, cv=kf)
    model.fit(X_train, y_train)
    test_preds[name] = model.predict(X_test)

meta = Ridge(alpha=1.0).fit(
    np.column_stack(list(oof.values())), y_train
)
y_pred = meta.predict(
    np.column_stack(list(test_preds.values()))
)
```

Common mistake: training the meta-learner on the base models' predictions from the *same* data the base models were trained on. That causes leakage: always use out-of-fold predictions (`cross_val_predict`), never `.predict()` on the training set.

12.5 Worked Example

Racing four ensemble methods on a synthetic house-price dataset gives a direct, measured comparison rather than a textbook ranking. The data has 2,000 houses and 14 features: square footage, room and bathroom counts, lot size, age, garage, pool, floors, distance to school, local crime rate, neighbourhood, condition, style, and heating type. The split is 80/20 train/test.

Listing 12.5 Training and benchmarking four ensemble methods

```
models = {
    'Random Forest': RandomForestRegressor(
        n_estimators=200, random_state=42, n_jobs=-1
    ),
    'Gradient Boosting': GradientBoostingRegressor(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=4,
        random_state=42,
    ),
    'XGBoost': xgb.XGBRegressor(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=4,
        random_state=42,
    ),
    'LightGBM': lgb.LGBMRegressor(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=4,
        random_state=42,
    ),
}
for name, model in models.items():
    model.fit(X_train, y_train)
    # time and score each model
```

Measured results:

Model	RMSE	R^2	Train time
Random Forest	\$45,502	0.8451	0.15s
Gradient Boosting	\$28,595	0.9388	0.47s

Model	RMSE	R^2	Train time
XGBoost	$28,873	0.9376	0.68s
LightGBM	$28,323	0.9400	**0.06s**

LightGBM is both the fastest to train (0.06s, roughly 2.5x faster than Random Forest and 8-11x faster than the other two boosted methods) and the most accurate here, edging out XGBoost's $28,873 and Gradient Boosting's $28,595. The "start with LightGBM" guidance earns its keep on both counts in this run, not just speed. Random Forest, with no boosting at all, is roughly $17,000 RMSE behind the boosted methods, a direct illustration of boosting's bias-reduction advantage over bagging on a task like this.

Feature importance (Random Forest) confirms `sqft` dominates as designed (0.551), with `neighborhood` a clear second (0.160). Below that the picture is less tidy than a designed dataset might suggest. `n_rooms` (0.065) and `school_dist` (0.060) both edge out `condition` (0.036), and `lot_size` (0.033) isn't negligible either, though the price formula weights it far more lightly than square footage. That's a useful reminder that a feature's importance to the model and its weight in the underlying formula aren't the same thing. Which features a tree-based model finds useful depends on how much separable signal they carry in this specific sample, not just how heavily they're weighted in the process that generated the data.

Sweeping `n_estimators` from 5 to 500 shows the two algorithms converging differently. LightGBM's RMSE drops sharply from $87,831 at 5 trees to $28,323 at 200 trees before flattening. Random Forest is already within a few thousand dollars of its final RMSE at 25 trees ($47,511 vs. $45,502 at 200). Bagging's variance reduction saturates fast, while boosting needs enough rounds to substantially reduce bias.

Finally, stacking a Random Forest and a LightGBM model with a Ridge meta-learner produced RMSE $29,535 ($R^2$ 0.9347). That's *worse* than solo LightGBM ($28,323), despite LightGBM being one of the two base models feeding the stack. The meta-learner's own weights explain why: Ridge assigns LightGBM a weight of 1.209 and Random Forest a weight of *-0.184*, meaning it learned to partly subtract Random Forest's much weaker predictions rather than blend them in. Even with that correction, including a clearly weaker model cost more than it gave back. This is a real, measured result, not a contrived counterexample. Stacking a strong model together with a much weaker one has no guarantee of beating the strong model alone. Procedure 12.4's "not a default choice" framing is exactly this finding in practice, not a hedge.

AWS SageMaker Equivalent

```
# XGBoost is a SageMaker built-in algorithm - no custom
# training script needed
import sagemaker
from sagemaker.estimator import Estimator

xgb_estimator = Estimator(
    image_uri=sagemaker.image_uris.retrieve(
        "xgboost", region, version="1.7-1"
    ),
    instance_type="ml.m5.xlarge",
    instance_count=1,
    hyperparameters={
        "num_round": 200, "max_depth": 4, "eta": 0.1
    },
)
xgb_estimator.fit({
    "train": train_s3_uri, "validation": val_s3_uri
})
```

When to prefer the built-in SageMaker XGBoost algorithm: when the team wants managed infrastructure, built-in distributed training for large datasets, and hyperparameter tuning jobs, without maintaining a custom training container. The trade-off is less flexibility than a hand-written `xgb.XGBRegressor` script for unusual preprocessing or custom objectives.

12.6 History and Further Reading

Murphy's Introduction, sections 18.1-18.4, gives the formal treatment of bagging, random forests, and boosting (Murphy 2022). Thakur, Ch. 6, covers practical gradient boosting with code (Thakur 2020). Burkov, Ch. 7, gives a crisp ensemble overview (Burkov 2019). Géron's *Hands-On Machine Learning*, Ch. 7, covers bagging, random forests, and boosting (AdaBoost and gradient boosting) together with the Scikit-Learn code for each (Géron 2019). The XGBoost paper is worth reading for the engineering behind its speed (Chen and Guestrin 2016).

12.7 How This Connects

This chapter builds on Chapter 11's decision trees as base learners. It looks ahead to Chapter 13's cross-validation, which is critical for stacking done correctly. It enables Chapter 14's feature importance work and Chapter 16's evaluation of ensemble models. It's often confused with neural networks as competing options for tabular data. Gradient boosting usually wins

on tabular data. Neural networks win on images, text, and audio, a distinction Book 2 returns to from Chapter 27 onward.

12.8 Exercises

1. [**W**] Explain in your own words why bagging reduces variance without increasing bias, using the σ^2/B relationship from this chapter's Exposition.
2. [**W**] State which of bagging or boosting each of the following targets: (a) a model that is systematically wrong on a specific subgroup of examples; (b) a model whose predictions change wildly with small changes to the training data.
3. [**A**] Using Listing 12.5's setup, explain why Random Forest's RMSE barely changes between `n_estimators=200` and `n_estimators=500` ($45,502 vs. $45,514) while LightGBM's RMSE at the same two settings keeps improving slightly, from $28,323 to $28,159. What does this suggest about each method's sensitivity to over-provisioning trees?
4. [**A**] Using Listing 12.4 as a base, add XGBoost as a third base model to the stack. State, from this chapter's own measured numbers, whether you'd expect the resulting stack to beat solo XGBoost's $28,873 RMSE.
5. [**E**] A customer's data science team reports "we stacked five models and it's more accurate than any of them individually." Using this chapter's Worked Example, write the two or three sentences you'd say before accepting that claim.
6. [**E**] A colleague argues Random Forest should always be tried before XGBoost "because it's simpler." Using Procedure 12.3 and this chapter's measured accuracy gap (Random Forest's RMSE roughly 58% higher than XGBoost's), write the case for when that advice is right and when it isn't.
7. [**R**] Read the XGBoost paper's (Chen & Guestrin, 2016) abstract and introduction. Identify one engineering optimisation it describes that this chapter's Trade-offs section does not mention. Explain what problem it solves.

12.9 Mental Model

A Random Forest asks: what do a hundred independent experts, each seeing a slightly different version of the data, collectively think? Gradient boosting asks: what's the smallest correction I can make to my current best guess? Stacking asks: which expert is right in which situation? But it still has to live with whichever experts were invited into the room. That's why including a much weaker expert, as this chapter's Worked Example did, can make stacking lose to its own best individual member.

13 Cross-Validation & Model Selection

A model trained and evaluated on a single 80/20 split reports one number. Let's say 94% accuracy. That number is less trustworthy than it looks. A different random split could easily give 91% or 97%. There's no way to tell from a single split which one you got. Now say twenty candidate models are compared on the same test set, and the best one is reported. That number is optimistic by construction. The best of twenty random-ish outcomes is not a neutral estimate. This chapter covers the systematic fix: cross-validation. It also covers the ways teams still leak information past it, even when they think they're doing it right.

13.1 Motivation

The basic rule: any data used to make decisions about a model cannot also be used to evaluate it. Choosing hyperparameters, selecting features, or picking between algorithms based on a test set uses that test set up. It's no longer a clean estimate of unseen-data performance. Cross-validation solves this. It repeatedly splits the data into different train/validation partitions and averages the results. No single split is "the answer". There is no best. There is only the distribution of performance across many splits.

13.2 Exposition

13.2.1 The Problem With a Single Train/Test Split

An 80/20 split on a 569-example dataset leaves only 114 examples to test on. That's enough for the estimate to swing a lot, depending on which 114 happened to land in the test set. With a small-to-moderate dataset, the standard error of a single accuracy estimate is large. A model can look like the best or worst result purely by chance.

13.2.2 k-Fold Cross-Validation

Split the data into k equal-sized folds. Train on $k-1$ folds. Validate on the remaining fold. Rotate which fold is held out. After k rounds every example has been in the validation set exactly once. Average the k scores. This uses every example for both training and validation.

The spread across folds, not just the mean, tells you how stable the model is. A mean with tight fold agreement is a different finding from the same mean with folds bouncing around. $k = 5$ is the standard default. $k = 10$ gives slightly lower bias (more training data per fold) at the cost of more compute. $k = n$, where n is the total number of examples in the dataset, is leave-one-out cross-validation (LOOCV). Each fold holds out exactly one example, trains on the remaining $n - 1$, and tests on that single point. Repeat n times, once per example, and average. LOOCV is unbiased in one sense: every training set is as large as it can possibly be. But it means fitting the model n separate times. And each fold's score is a single right-or-wrong outcome, with no averaging within the fold itself. That's where the Trade-offs table's "high variance between folds" comes from. LOOCV is practical for the tiny datasets it's usually reserved for. Fitting a model 500 times, leaving out one of 500 rows each time, is a very different thing on a larger dataset.

13.2.3 Stratified k-Fold

Regular k-fold can, by chance, produce folds with very different class ratios. If 5% of the data is the positive class, one fold could land at 0% positives, making that fold's score incomparable to the others. Stratified k-fold preserves the class ratio in every fold, and is the correct default for classification; `cross_val_score` uses `StratifiedKFold` automatically when the estimator is a classifier.

13.2.4 Time-Series Cross-Validation

Random shuffling is wrong for time-ordered data. Training on next week's data to predict this week's leaks future information into the training set. The result is an optimistic estimate that collapses the moment it meets production. `TimeSeriesSplit` enforces time order. The validation set is always in the future relative to its training set. Training windows grow across folds. Validation windows stay strictly ahead of them. This is a strict requirement for any time-ordered dataset: sales, prices, sensor readings, user behaviour over time.

13.2.5 The Overfitting-to-Test-Set Problem

A common, easy-to-fall-into failure mode. Train fifty different models. Measure each on the same test set. Report the best one's test accuracy as "the" model's performance. Selecting the maximum over fifty imperfect estimates introduces selection bias. The winning number is optimistic. Not because the winning model is secretly worse, but because the *selection process itself* systematically favours whichever model got lucky on that specific test set.

13.2.6 Nested Cross-Validation

Nested CV separates model *selection* from model *evaluation* using two loops. An outer loop (typically $k = 5$) estimates generalisation performance. An inner loop (typically $k = 3$, runs within each outer training set) selects the best hyperparameters. For each outer fold, run the inner-loop hyperparameter search on that fold's training data. Then evaluate the resulting best model on that fold's held-out test data, a test set the inner search never touched. The hyperparameters chosen may differ across outer folds. That's expected. The outer loop is estimating expected performance. It is not selecting the model to ship.

13.3 Trade-offs and Limits

CV Strategy	When to use	Weakness
Single train/test split	Quick sanity checks only	High variance, unreliable
k-Fold (k=5)	Standard default for most tasks	May lose temporal ordering
Stratified k-Fold	Classification with class imbalance	Same as k-Fold otherwise
TimeSeriesSplit	Any time-ordered data	Training set smaller in early folds
Nested CV	Hyperparameter tuning + honest evaluation	Computationally expensive
Leave-One-Out	Tiny datasets (<50 examples)	Slow; high variance between folds

13.4 Procedures

Procedure 13.1 (Run k-Fold CV in One Line). *When to use:* estimating generalisation performance of any model.

Listing 13.1 Standard k-fold CV inside a Pipeline

```
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression(max_iter=1000)),
])
scores = cross_val_score(pipe, X, y, cv=5, scoring='accuracy')
print(f'Accuracy: {scores.mean():.3f} ± {scores.std():.3f}')
```

Common mistake: fitting a scaler (or any preprocessing) on the whole dataset before cross-validating, rather than inside a `Pipeline`. That leaks validation-fold statistics into training and produces optimistic scores. Always put preprocessing inside the pipeline being cross-validated.

Procedure 13.2 (Use Stratified k-Fold Explicitly). *When to use:* imbalanced classification, or when explicit control over the CV strategy is needed.

Listing 13.2 Explicit stratified k-fold

```
from sklearn.model_selection import (
    StratifiedKFold,
    cross_val_score,
)

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(
    pipe, X, y, cv=skf, scoring='f1_weighted'
)
```

Procedure 13.3 (Set Up Time-Series CV). *When to use:* any dataset where rows are time-ordered.

Listing 13.3 TimeSeriesSplit: validation always in the future

```
from sklearn.model_selection import (
    TimeSeriesSplit,
    cross_val_score,
)

# Verify chronological order first
df = df.sort_values('date').reset_index(drop=True)
tss = TimeSeriesSplit(n_splits=5)
scores = cross_val_score(
    pipe, X_ts, y_ts, cv=tss, scoring='accuracy'
)
```

Procedure 13.4 (Implement Nested CV for Honest Hyperparameter Tuning). *When to use:* tuning hyperparameters and reporting a valid performance estimate at the same time.

Listing 13.4 Nested cross-validation: inner search, outer honest evaluation

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import (
    GridSearchCV,
    cross_val_score,
    StratifiedKFold,
)

rf_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', RandomForestClassifier(random_state=42)),
])

inner_cv = StratifiedKFold(
    n_splits=3, shuffle=True, random_state=42
)
outer_cv = StratifiedKFold(
    n_splits=5, shuffle=True, random_state=42
)

param_grid = {
    'clf__n_estimators': [50, 100, 200],
    'clf__max_depth': [3, 5, None],
}
search = GridSearchCV(
```

```
    rf_pipe, param_grid, cv=inner_cv, scoring='accuracy', n_jobs=-1
)

nested_scores = cross_val_score(
    search, X, y, cv=outer_cv, scoring='accuracy'
)
mean, std = nested_scores.mean(), nested_scores.std()
print(f'Nested CV: {mean:.3f} ± {std:.3f}')

# After nested CV establishes the honest estimate, refit
# on all data for the final model:
search.fit(X, y)
final_model = search.best_estimator_
```

Procedure 13.5 (Compare Two Models Properly). *When to use:* before claiming "Model A is better than Model B."

Listing 13.5 Paired t-test on cross-validation fold scores

```
from scipy import stats

scores_a = cross_val_score(
    model_a, X, y, cv=10, scoring='accuracy'
)
scores_b = cross_val_score(
    model_b, X, y, cv=10, scoring='accuracy'
)
t_stat, p_val = stats.ttest_rel(scores_a, scores_b)
```

Common mistake: comparing two models' mean CV scores directly without a significance test. A 0.3 percentage-point difference in means can easily be noise. A paired t-test on matched fold scores is the quick check. Treat it as optimistic: the folds share training data, so their scores aren't independent, and a plain t-test overstates confidence. Nadeau and Bengio's corrected resampled t-test adjusts for that.

13.5 Worked Example

Take the Breast Cancer Wisconsin dataset: 569 examples, binary classification. Run the same logistic-regression pipeline through a fresh 80/20 split under ten different random seeds. Accuracy ranges from 0.965 to 1.000, a 3.5 percentage-point spread from nothing but which examples happened to land in the test set:

Listing 13.6 Ten random splits, same model, same data: different reported accuracy

```
for seed in range(10):
    X_tr, X_te, y_tr, y_te = train_test_split(
        X, y, test_size=0.2, random_state=seed, stratify=y
    )
    pipe.fit(X_tr, y_tr)
    acc = accuracy_score(y_te, pipe.predict(X_te))
    print(f'Seed {seed}: {acc:.3f}')
```

Any single one of those ten numbers, reported alone, would have looked like a clean, confident result. Run a 5-fold stratified CV instead, on the same pipeline and data. It reports fold scores of [0.982, 0.982, 0.974, 0.974, 0.991], mean 0.981, std 0.007. That's a single number with an honest error bar attached, built from the same data used five different ways rather than one lucky (or unlucky) way.

Nested CV tunes a Random Forest's `n_estimators`/`max_depth` via a 3-fold inner search inside a 5-fold outer loop. The result here is more subtle than the "nested is always dramatically lower" framing sometimes suggests. The non-nested estimate (the search's own `best_score_`, from the same folds used to pick the winning hyperparameters) reports 0.9543. The honest nested estimate reports 0.9543 too. That's a gap too small to separate from rounding on this particular dataset, not the large optimism bias the Overfitting-to-Test-Set Problem describes in the worst case. That's still a real, useful finding in its own right. The size of the nested-vs-non-nested gap depends on how easy the dataset is to separate and how much hyperparameter search space was explored. The only way to know whether a given case is a small gap or a large one is to run nested CV, not to assume.

AWS SageMaker Equivalent

```
# SageMaker Automatic Model Tuning runs the equivalent of a
# hyperparameter search with built-in objective-metric
# tracking across training jobs
import sagemaker
from sagemaker.tuner import (
    HyperparameterTuner,
    IntegerParameter,
)

tuner = HyperparameterTuner(
    estimator=xgb_estimator,
    objective_metric_name="validation:accuracy",
    hyperparameter_ranges={
        "max_depth": IntegerParameter(3, 10)
    },
```

```
    max_jobs=20,
    max_parallel_jobs=4,
)
tuner.fit({
    "train": train_s3_uri,
    "validation": val_s3_uri,
})
```

When to prefer SageMaker Automatic Model Tuning: when the search needs to run as many independent, parallel managed training jobs rather than in-process `GridSearchCV`. Useful once a single search no longer fits comfortably on one machine, or when the team wants tuning history tracked centrally rather than in a notebook.

13.6 History and Further Reading

Thakur, Ch. 2, gives practical cross-validation patterns with code (Thakur 2020). Murphy's Introduction, sections 4.5-4.6, covers the statistical foundations of model selection (Murphy 2022). Burkov, Ch. 5, covers hyperparameter tuning and model selection (Burkov 2019).

13.7 How This Connects

This chapter builds on Chapter 11 (a model is needed before it can be cross-validated). It looks ahead to Chapter 16 for the evaluation metrics used within CV. It enables Chapter 17's hyperparameter optimisation, which is always done via CV. It's needed for essentially every model-comparison decision in the rest of this book. It's often confused with test-set evaluation. CV is for model development and selection. The test set is for one final evaluation, after all decisions have already been made. Touch the test set once.

13.8 Exercises

1. **[W]** Explain in your own words why choosing hyperparameters based on test-set performance invalidates that test set as an estimate of generalisation.
2. **[W]** State which CV strategy is correct for each: (a) a dataset of 30 rare-disease patient records; (b) daily sales figures for the past three years; (c) a roughly balanced 5,000-row classification dataset.

3. **[A]** Using Listing 13.6's pattern, compute the standard deviation of the ten seed-accuracy values (0.982, 0.991, 0.982, 0.974, 0.974, 0.982, 0.991, 0.965, 0.974, 1.000) and compare it to this chapter's 5-fold CV standard deviation (0.007). Which estimate would you trust more, and why?
4. **[A]** Using Listing 13.4, explain what it would mean if the non-nested and nested CV estimates came back nearly identical (as this chapter's own 0.9543 vs. 0.9543 result did) versus if they came back several percentage points apart.
5. **[E]** A teammate says "I ran 5-fold CV and got 94% ± 0.5%, so I'm confident in this number." Name one thing that CV std alone does not tell you about the estimate's reliability.
6. **[E]** A customer has a time-ordered dataset but their data science team used regular `StratifiedKFold` and reported a strong result. Using this chapter's Time-Series Cross-Validation section, write the two or three sentences you'd say about why that result should not be trusted as-is.
7. **[R]** Read the scikit-learn documentation's page on `cross_validate` versus `cross_val_score`. Identify one piece of information `cross_validate` returns that `cross_val_score` does not, and describe a situation from this chapter where that extra information would matter.

13.9 Mental Model

Cross-validation is not a model selection technique. It's an estimation technique. It tells you how well your model *type* generalises. The test set tells you how well *this specific fitted model* generalises. Use CV during development. Touch the test set once, at the end, to report the final result. Never let it take part in any decision along the way.

14 Feature Engineering

Cross-validation is set up correctly. A bunch of algorithms have been tried. Accuracy is stuck. Nothing gets past a mediocre ceiling. The bottleneck isn't the model. It's the features. A model can only work with the signal it's given. Say the information needed to make a good prediction is buried in the raw data, in a form the model can't see. No amount of algorithmic cleverness closes that gap. This chapter is about making that hidden information visible.

14.1 Motivation

A feature is a measurable property of an observation that a model uses to predict. Feature engineering transforms raw data into features that make patterns easier to learn. The question to ask of every column: what information is latent in this data that the model currently can't see? A datetime column contains day-of-week, time-of-day, holiday status, and recency, all at once. An address contains a city, a region, a distance to a point of interest. A text description contains sentiment, keywords, length. Feature engineering extracts these hidden signals and presents them explicitly. It doesn't leave the model to guess at them from a single opaque column.

14.2 Exposition

14.2.1 Numeric Feature Transformations

Scaling matters for linear and distance-based models (KNN, SVM), which are sensitive to feature magnitude. A feature measured in thousands (income) will dominate one measured in single digits (age) unless scaled. `StandardScaler` (subtract mean, divide by std) is the default choice. `MinMaxScaler` (scale to $[0, 1]$) suits bounded-input needs like neural networks. `RobustScaler` (median and IQR instead of mean and std) suits data with real outliers. Tree-based models (Random Forest, XGBoost) split on thresholds, not magnitudes. They need no scaling at all.

Log transforms compress the right tail of right-skewed distributions (income, price, population), where a few very large values drag the mean far from the median. `np.log1p` (which

handles zeros safely) gives the model a more balanced signal, and is worth reaching for whenever a histogram of the feature looks like an exponential curve.

Listing 14.1 Right-skewed income data, before and after a log transform

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.RandomState(42)
income = rng.lognormal(mean=10.5, sigma=0.6, size=2000)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].hist(income, bins=40, color='#4E79A7')
axes[0].axvline(income.mean(), color='#999999', linestyle='--', label='mean')
axes[0].axvline(np.median(income), color='#4E79A7', linestyle=':',
                linewidth=2.5, label='median')
axes[0].set_title('Raw income: mean pulled right of the median')
axes[0].legend()

log_income = np.log1p(income)
axes[1].hist(log_income, bins=40, color='#59A14F')
axes[1].axvline(log_income.mean(), color='#999999', linestyle='--', label='mean')
axes[1].axvline(np.median(log_income), color='#4E79A7', linestyle=':',
                linewidth=2.5, label='median')
axes[1].set_title('log1p(income): mean and median nearly coincide')
axes[1].legend()
plt.tight_layout()
plt.show()
```

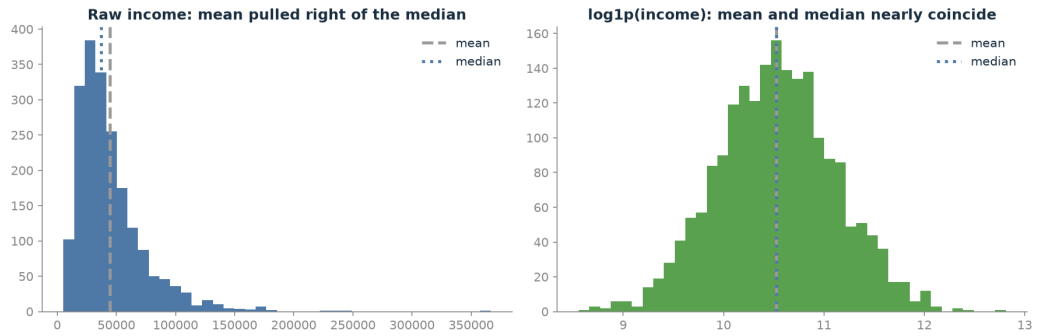


Figure 14.1: Synthetic lognormal income data (n=2,000), before and after a log1p transform.

Figure 14.1 shows exactly the pattern the maths predicts. Raw income has mean 44,543 sitting well above median 37,302, the long right tail pulling the average away from where

most of the mass is. The log-transformed version has mean 10.5271 against median 10.5268, close enough to call the same number. That's the visual signature of a distribution the model can now treat as roughly symmetric.

Polynomial features let linear models capture non-linear relationships they otherwise can't represent. If y depends on X quadratically (a sweet spot where too little or too much of X is bad), adding X^2 explicitly via `PolynomialFeatures` is what lets a linear model see that curvature.

Listing 14.2 A quadratic relationship: unfit by a line on X, fit by a line on (X-1)-squared

```
X = rng.uniform(-5, 5, 300)
y = -(X - 1) ** 2 + 10 + rng.normal(0, 1.5, 300)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].scatter(X, y, alpha=0.5, color='#4E79A7')
axes[0].set_xlabel('X')
axes[0].set_ylabel('y')
axes[0].set_title('y vs. X: a straight line can\'t follow this curve')

axes[1].scatter((X - 1) ** 2, y, alpha=0.5, color='#59A14F')
axes[1].set_xlabel('(X-1)²')
axes[1].set_ylabel('y')
axes[1].set_title('y vs. (X-1)²: now a straight line fits')
plt.tight_layout()
plt.show()
```

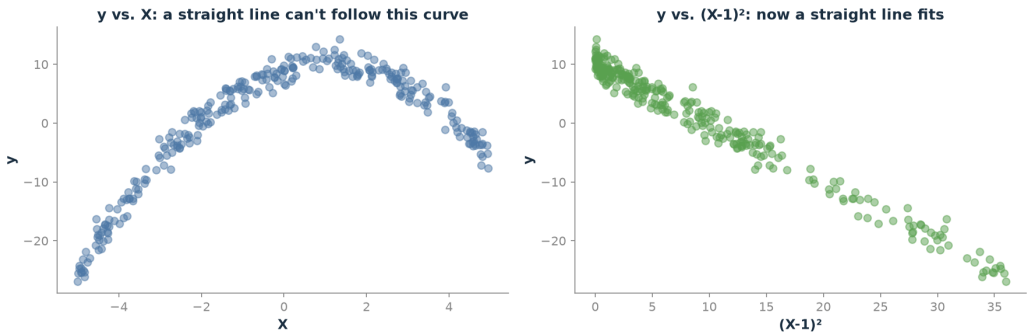


Figure 14.2: y depends on X quadratically, with a sweet spot near X=1. A straight line can't follow the left panel; the same data, re-expressed on (X-1)², is nearly linear.

Figure 14.2 is the same curvature a linear model sees when it's only given X. Whatever line it draws, it's wrong at both ends or wrong in the middle. The correlation between X and y

measures a modest 0.62. The right panel is the identical data, re-expressed on $(X-1)^2$. The curve has been straightened out entirely by the feature, not by the model: correlation jumps to -0.99, close to the -1 of a perfectly linear relationship (negative, since further from the sweet spot at $X = 1$ means lower y). That's the whole point of adding the feature explicitly, rather than hoping a linear model discovers the relationship on its own.

Binning converts a continuous variable into discrete categories, useful when the true relationship is non-linear but step-like (age groups in marketing, income brackets in insurance) via `pd.cut` or `KBinsDiscretizer`.

14.2.2 Categorical Feature Encoding

Models need numbers, so categorical variables must be converted. **One-hot encoding** creates a binary column per category. It's best for nominal categories (no natural order) with low cardinality (under roughly 20 unique values). **Label/ordinal encoding** assigns an integer per category. This is only safe when the categories have a genuine order (poor < fair < good < excellent), or when the downstream model is tree-based and won't be misled by a false ordering. A linear model fed label-encoded nominal categories will treat category 2 as "twice" category 1, an ordering that doesn't exist. **Target encoding** replaces each category with the mean of the target for that category. It handles high cardinality (postal codes, product IDs, user IDs) where one-hot would explode into thousands of columns. But it leaks label information if done naively. It must always be computed with fold-holdout cross-validation, never a single global mean. **Frequency encoding** replaces each category with how often it appears. It never looks at the label, so it carries no label leakage, and captures the idea that rare categories may behave differently from common ones.

14.2.3 Datetime Feature Extraction

A single datetime column hides hour, day-of-week, month, year, weekend/holiday status, and days-since-a-reference-point. All extractable directly via `.dt` accessors. For cyclical features (hour, day of week, month), the raw integer gets the distance wrong: hour 23 and hour 0 are one hour apart in reality but 23 apart numerically. Cyclical (sin/cos) encoding fixes this by mapping the cycle onto a circle, so adjacent-in-time values stay numerically adjacent.

Listing 14.3 24 hours mapped onto a circle via sin/cos: hour 23 sits next to hour 0

```
fig, axes = plt.subplots(1, 2, figsize=(11, 5))

hours = np.arange(24)
axes[0].scatter(hours, np.zeros_like(hours), c=hours, cmap='twilight', s=60)
for h in [0, 6, 12, 18, 23]:
    axes[0].annotate(str(h), (h, 0), textcoords='offset points', xytext=(0, 10))
axes[0].set_yticks([])
axes[0].set_xlabel('raw hour')
axes[0].set_title('Raw hour: 23 and 0 sit at opposite ends')
axes[0].spines['left'].set_visible(False)

hour_frac = 2 * np.pi * hours / 24
hour_sin, hour_cos = np.sin(hour_frac), np.cos(hour_frac)
axes[1].scatter(hour_cos, hour_sin, c=hours, cmap='twilight', s=60)
for h in [0, 6, 12, 18, 23]:
    axes[1].annotate(str(h), (hour_cos[h], hour_sin[h]),
                     textcoords='offset points', xytext=(5, 5))
axes[1].set_xlabel('hour_cos')
axes[1].set_ylabel('hour_sin')
axes[1].set_title('sin/cos hour: 23 sits right next to 0')
axes[1].set_aspect('equal')
plt.tight_layout()
plt.show()
```

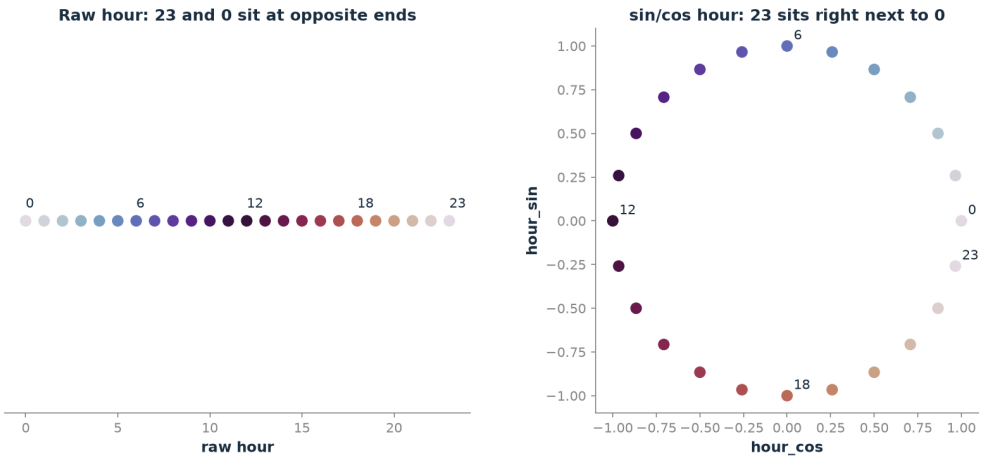


Figure 14.3: All 24 hours, plotted by raw value (left) and by (hour_cos, hour_sin) coordinates (right).

On the left of Figure 14.3, the raw hour axis puts 23 and 0 as far apart as two numbers can be, even though midnight to 11pm is one hour. On the right, the same 24 hours plotted as `(hour_cos, hour_sin)` coordinates fall on a circle. 23 lands right beside 0, exactly the adjacency the raw integer got wrong.

14.2.4 Interaction Features

An interaction feature is the product or ratio of two features. It captures a joint effect neither captures alone. Square metres and neighbourhood are each important for house price, but their *interaction* is what matters. A large house in an expensive neighbourhood is worth far more than the sum of a generic "large house" premium and a generic "good neighbourhood" premium. Tree-based models implicitly build interactions through their splits, so explicit interaction features matter less for them. For linear models, they're often essential to capture non-additive effects at all.

14.2.5 Domain-Specific Features

The highest-leverage features usually come from domain knowledge, not generic transformations. For churn prediction, Recency (days since last purchase), Frequency (purchases in the last 90 days), and Monetary value (RFM, a decades-old direct-marketing framework) often outperform raw transaction data. They capture what drives churn, rather than a generic statistical summary of it. The best feature engineers spend the time talking to domain experts before writing code.

> **i** What Is RFM?
>
> RFM scores every customer on three numbers, each computed straight from transaction history: **Recency**, days since their last purchase; **Frequency**, how many purchases they've made in a fixed window; and **Monetary**, how much they've spent in that window. The framework predates machine learning by decades. Direct-mail marketers used it by hand, ranking customers into quintiles on each dimension and mailing only to the highest-scoring segments. Postage cost money, and a badly-targeted catalogue was a real loss, not just a wasted API call (Hughes 1994). As three engineered features, RFM works for the same reason it worked on paper: a customer who bought recently, buys often, and spends a lot is a very different prospect from one who did all three once, years ago. Each of the three numbers on its own is a weaker signal than the combination. Recency alone doesn't distinguish a big spender from a one-off. Frequency alone doesn't distinguish someone buying small items constantly from someone who checked in twice with two large orders. It's a domain-knowledge feature in the purest sense. Nothing about the transaction table forces this particular three-number summary to be the useful one. It took decades of direct-marketing practice to establish that it is.

14.3 Trade-offs and Limits

More features raise overfitting risk. Each addition should be validated with cross-validation. Feature engineering is also a leakage risk. Any feature using information from the future, or from the test set, produces optimistic training results that fail in production. Target encoding is the pointiest version of this trap. It's easy to compute correct-looking statistics on the wrong data. Scaling is part of the preprocessing pipeline, fit on training data only, then applied (never re-fit) to test data.

14.4 Procedures

Procedure 14.1 (Choose Between One-Hot, Ordinal, and Target Encoding). *When to use:* deciding how to encode any categorical column.

Listing 14.4 Encoder choice by cardinality and order

```
# One-hot: nominal, low cardinality (< ~20)
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(
    handle_unknown='ignore', sparse_output=False
)

# Ordinal: genuine order (poor < fair < good < excellent)
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder(
    categories=[['poor', 'fair', 'good', 'excellent']]
)

# Target/mean encoding for high cardinality: ALWAYS via
# CV-encoding, never a single global mean
```

Common mistake: using integer label encoding for a nominal column feeding a linear model. The model will read category 2 as "twice" category 1.

Procedure 14.2 (Target-Encode Without Leaking the Label). *When to use:* a high-cardinality categorical where frequency encoding isn't informative enough.

Listing 14.5 Leakage-safe target encoding via fold-holdout

```
def cv_target_encode(
    X_train, y_train, X_test, column, n_splits=5, smoothing=10
):
    global_mean = y_train.mean()
    train_encoded = pd.Series(index=X_train.index, dtype=float)
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    for tr_idx, val_idx in kf.split(X_train):
        fold_train = X_train.iloc[tr_idx]
        fold_y = y_train.iloc[tr_idx]
        stats = fold_y.groupby(fold_train[column]).agg(
            ['mean', 'count']
        )
        smoothed = (
            stats['mean'] * stats['count']
            + global_mean * smoothing
        ) / (stats['count'] + smoothing)
        val_rows = X_train.iloc[val_idx][column]
        train_encoded.iloc[val_idx] = (
            val_rows.map(smoothed).fillna(global_mean).values
        )
    full_stats = y_train.groupby(X_train[column]).agg(
        ['mean', 'count']
    )
    full_smoothed = (
        full_stats['mean'] * full_stats['count']
        + global_mean * smoothing
    ) / (full_stats['count'] + smoothing)
    test_encoded = X_test[column].map(full_smoothed).fillna(
        global_mean
    )
    return train_encoded, test_encoded
```

Common mistake: computing `df.groupby(column)[target].mean()` on the full dataset before splitting, then mapping it onto both train and test. This bakes test-set target information into every training row.

Procedure 14.3 (Extract Features from a Datetime Column). *When to use:* any timestamp column.

Listing 14.6 Datetime extraction with cyclical encoding

```
def add_datetime_features(df, col):
    df = df.copy()
    df[f'{col}_hour'] = df[col].dt.hour
    df[f'{col}_dow'] = df[col].dt.dayofweek
    is_weekend = df[col].dt.dayofweek.isin([5, 6])
    df[f'{col}_is_weekend'] = is_weekend.astype(int)
    hour_frac = 2 * np.pi * df[f'{col}_hour'] / 24
    df[f'{col}_hour_sin'] = np.sin(hour_frac)
    df[f'{col}_hour_cos'] = np.cos(hour_frac)
    return df
```

Procedure 14.4 (Build a Leakage-Safe `ColumnTransformer` Pipeline). *When to use:* a mix of numeric and categorical columns that need to be cross-validated safely as one unit.

Listing 14.7 ColumnTransformer wrapping numeric and categorical preprocessing

```
numeric_pipe = Pipeline([
    ('impute', SimpleImputer(strategy='median')),
    ('scale', StandardScaler()),
])
categorical_pipe = Pipeline([
    ('impute', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore')),
])
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_pipe, numeric_features),
    ('cat', categorical_pipe, categorical_features),
])
full_pipeline = Pipeline([
    ('preprocess', preprocessor),
    ('model', your_model),
])
```

Common mistake: calling `scaler.fit_transform(X)` on the whole dataset before `train_test_split`. Split first, `.fit()` on train only, `.transform()` (never `.fit_transform()`) on test.

14.5 Worked Example

A synthetic e-commerce order dataset (4,000 orders; features include customer ID, region, payment method, age, items in cart, and an order timestamp; 3,200/800 train/test split) makes the leakage question concrete rather than abstract. Encode the high-cardinality `customer_id` column two ways: a naive global mean computed across the *whole* dataset before splitting, versus Procedure 14.2's fold-holdout CV encoding. Then compare cross-validated R^2 on the training set alone:

Listing 14.8 Leaky vs. CV-safe target encoding, measured

```
scores_leaky = cross_val_score(
    ridge, X_train_leaky[feat_cols], y_train, cv=5, scoring='r2'
)
scores_safe = cross_val_score(
    ridge, X_train_safe[feat_cols], y_train, cv=5, scoring='r2'
)
```

Measured: leaky encoding reports CV $R^2 = 0.872 \pm 0.009$. Safe encoding reports 0.870 ± 0.010. Leaky encoding looks very slightly better, exactly the direction the leakage warning predicts, even though the gap here is modest. Checking the *true* held-out test R^2 for both confirms the honest picture: leaky encoding scores 0.870 on test, safe encoding scores 0.867, nearly identical to their respective CV numbers. Why so close? In this particular synthetic dataset, `customer_id` isn't a strongly predictive column on its own. Order value is driven mostly by region, items in cart, and weekend timing, not by which customer placed the order. The lesson this measurement teaches is more precise than "leakage always inflates your score dramatically." Leakage's damage scales with how much the leaking column correlates with the target once leaked. A weakly-predictive ID column leaks only a little. A strongly-predictive one (e.g. a column that indirectly encodes the label) can produce the dramatic, disqualifying gap the warnings describe. Measuring the gap, rather than assuming its size, is the only way to know which situation is in play.

The full pipeline (cyclical datetime features, one-hot encoded region and payment method, no customer ID at all) reaches CV $R^2 = 0.870 \pm 0.010$ and true test $R^2 = 0.867$. That's identical to the safely-encoded variant above, and only a hair behind the leaky one. That tie is the real finding. Honestly encoding `customer_id` costs nothing next to dropping it outright, because the column carries no signal a fold-holdout encoding can safely pull out. The leaky version's tiny edge comes entirely from peeking at the label, not from `customer_id` actually mattering.

AWS SageMaker Equivalent

```
# SageMaker Data Wrangler: no-code equivalent for scaling,
# encoding, and datetime extraction via a visual flow,
# exportable as a Processing job.

import sagemaker
from sagemaker.feature_store.feature_group import FeatureGroup

feature_group = FeatureGroup(
    name='orders-features',
    sagemaker_session=sagemaker.Session(),
)
# feature_group.create(
#     s3_uri='s3://my-bucket/feature-store/orders/',
#     record_identifier_name='order_id',
#     event_time_feature_name='event_time',
#     role_arn=role, enable_online_store=True)
```

When to prefer SageMaker Feature Store: when the same engineered feature (e.g. a customer's frequency encoding) needs to be computed once and served consistently to both a training job and a real-time inference endpoint, preventing training/serving skew.

14.6 History and Further Reading

Thakur, Ch. 4 and 5, cover categorical encoding and feature engineering for various data types with code (Thakur 2020). Murphy's Introduction, Ch. 9, gives the formal treatment of linear models and feature maps (Murphy 2022). Zheng and Casari's *Feature Engineering for Machine Learning* is a full book on the topic (Zheng and Casari 2018).

14.7 How This Connects

This chapter builds on Chapter 7 (clean data before engineering features) and Chapter 9 (EDA is usually where the need for a given transformation is discovered). It enables Chapter 15's feature selection, which needs features to exist before choosing among them. It also enables Chapter 16's model evaluation, which is what any feature engineering decision gets judged against in the end. It's often confused with feature *selection*: engineering *creates* new features, while selection *chooses* which existing features to keep.

14.8 Exercises

1. [**W**] Explain why tree-based models don't need feature scaling while linear and distance-based models do, using the "splits on thresholds, not magnitudes" framing from this chapter.
2. [**W**] For each, name the correct encoding from this chapter: (a) a `size` column with values `small`/`medium`/`large`; (b) a `product_sku` column with 50,000 unique values feeding a linear model; (c) a `country` column with 15 unique values, no natural order.
3. [**A**] Using Listing 14.6, compute `hour_sin`/`hour_cos` for hour 23 and hour 0 by hand, and show numerically that their Euclidean distance is small, in contrast to the raw hour values, which differ by 23.
4. [**A**] Using Listing 14.8's pattern, explain what result you would expect if `customer_id` were replaced with a column that directly encoded a noisy version of the label (e.g. `last_order_value_rounded`), would the leaky-vs-safe CV R^2 gap this chapter measured (0.872 vs. 0.870) likely grow, shrink, or stay about the same?
5. [**E**] A teammate wants to one-hot encode a `product_id` column with 8,000 unique values "to keep things simple." Using this chapter's cardinality guidance, write the two or three sentences you'd say instead.
6. [**E**] This chapter's measured leaky-vs-safe gap was small (0.872 vs. 0.870). Using the Worked Example's explanation for why, describe a realistic feature (not `customer_id`) in the same order dataset that would very likely produce a much larger leakage gap if encoded naively.
7. [**R**] Read the scikit-learn `TargetEncoder` documentation (or `category_encoders.TargetEncoder`). Identify one detail of its built-in cross-fitting behaviour that differs from Procedure 14.2's hand-written version, and explain what problem that difference solves.

14.9 Mental Model

Features are the vocabulary a model uses to describe an observation. The richer and more precise that vocabulary, the better the model can describe the pattern it's trying to learn. Feature engineering is translation work. Taking data in one language (raw measurements) and translating it into the language the model speaks. The numbers that carry signal.

15 Feature Selection & Dimensionality Reduction

Feature engineering fills a dataset with columns. This chapter is about admitting that not all of them are useful. One-hot encoding, interaction terms, datetime extraction: you've done it all. A modelling table can carry hundreds of columns. Past a point, more columns stop helping and start hurting. Models overfit. Training slows. Distance-based methods stop behaving sensibly. Two different operations both go by the name "dimensionality reduction," which invites confusion rather than avoiding it. Feature *selection* keeps a subset of the original columns, so "income" stays income. Feature *extraction* builds a smaller number of new columns that are blends of the originals, trading interpretability for compactness. Both exist to fight the same enemy: the curse of dimensionality. Knowing which one a given situation needs matters as much as knowing how to run either.

15.1 Motivation

Chapter 14's feature engineering deliberately produces more candidate columns than a model needs. The theory is that it's cheaper to generate too much signal and prune it than to under-engineer and miss something. That pruning step is this chapter. Without it, a model trained on hundreds of largely redundant or irrelevant features doesn't just run slower. It performs worse. It has more surface area on which to fit noise instead of signal. The goal is a feature set that is as small as it can be, without discarding anything that helps. Get there by keeping the best of what already exists (selection), or by compressing everything into fewer, denser dimensions (extraction).

15.2 Exposition

15.2.1 The Curse of Dimensionality

Three things get worse as the number of features grows. *Distances stop being informative.* In high dimensions, the ratio between the nearest and farthest neighbour's distance shrinks towards 1. "Nearest" stops meaning much for the algorithms that depend on it: KNN, K-Means, anything built on Euclidean or cosine similarity. *The data becomes sparse relative*

to the space. Keeping the same density of training examples per unit volume requires the number of examples to grow exponentially with the number of dimensions. With 20 features and a modest sample size, the data is a thin cloud in a mostly empty 20-dimensional space. *Overfitting risk rises.* More features means more ways to fit noise rather than signal, especially with a fixed number of rows. 500 features and 1,000 rows gives a model enormous freedom to memorise the training set rather than learn from it. There's also a practical cost, independent of any statistical concerns: more features means more compute, more memory, and slower training.

The sparsity claim is easy to say and easy to underestimate, so it's worth making concrete with numbers. Picture a cube of jelly, one unit of length on each side, with a million tiny balls suspended in it. One ball per data point. Say the cube has 3 dimensions: three features, each ranging over that unit of length. A million evenly-spaced balls sit roughly $1/\sqrt[3]{1{,}000{,}000} = 1/100$ of the cube's side apart from their nearest neighbours. That's close enough that "nearest" means something. Now stretch that same cube into 20 dimensions. One axis per feature. Keep the same million balls. The average spacing between a ball and its nearest neighbour becomes $1/\sqrt[20]{1{,}000{,}000} \approx 1/1.995 \approx 0.501$: roughly half the entire length of the cube's side. The same million points that felt dense in 3 dimensions are, in 20 dimensions, spread so thinly that each point's nearest neighbour is half the cube away. No amount of data collection closes that gap. Matching the 3-dimensional spacing of 1/100 in 20 dimensions would need $100^{20} = 10^{40}$ balls, about 10^{34} times the million we started with. This is the curse. Adding dimensions doesn't make a dataset a little sparser. It makes it exponentially, catastrophically sparser, for a fixed number of examples.

15.2.2 Feature Selection: Filter, Wrapper, Embedded

Selection methods fall into three families, distinguished by how they decide which features matter.

Filter methods score each feature independently of any model, then keep the top scorers:

- **Variance threshold** drops near-constant features, which carry no signal for any model.
- **Correlation with the target** keeps features most linearly related to `y`. It's cheap but blind to non-linear relationships, so a feature with a perfect U-shaped relationship to the target scores near-zero linear correlation and gets wrongly discarded.
- **Mutual information** measures any dependency, linear or not, between a feature and the target. Slower than correlation, but far more reliable at catching non-linear signal.

Filters are fast (score everything once, sort, cut) but they ignore how features interact with each other and with whatever model will consume them.

Wrapper methods use the downstream model's own performance to decide what to keep, searching over subsets directly rather than scoring features in isolation. Recursive Feature

Elimination (RFE) trains the model, ranks features by importance or coefficient magnitude, drops the weakest, and repeats until reaching a target count. This directly optimises for what matters: model performance. But it's expensive. It retrains the model repeatedly. And the resulting subset is specific to whichever model did the selecting.

Embedded methods get selection for free as a side effect of training a model that has sparsity or importance built in. L1 regularisation (Lasso) drives some coefficients exactly to zero, building selection into the loss function (Chapter 18 covers regularisation in depth). Tree-based importances (Random Forest, gradient-boosted trees) naturally produce a feature ranking as a byproduct of how often and how effectively each feature was used for splitting. Embedded methods are usually the best default in practice: they give selection signal without a separate search loop. But the importance scores can be biased towards high-cardinality or correlated features. Treat them as a hint, nothing more.

15.2.3 Feature Extraction: PCA

Principal Component Analysis doesn't select features. It constructs new ones. The idea: find the direction in feature space along which the data varies most, call that the first principal component. Find the next direction (orthogonal to the first) that captures the most *remaining* variance. Repeat. Mathematically this is an eigendecomposition of the feature covariance matrix. If Σ is the covariance matrix of the standardised features, PCA finds its eigenvectors and eigenvalues:

$$\Sigma v = \lambda v$$

Each eigenvector v is a principal component (a direction in feature space) and its eigenvalue λ is the amount of variance captured along that direction. Sorting components by eigenvalue, largest first, and keeping the top k gives the k-dimensional linear subspace that preserves the most variance possible for that value of k. In practice, a dataset almost never needs every original dimension to explain the bulk of its variance. Ten correlated features might contain only three or four independent "directions" of information. PCA finds them.

Standardise features before applying PCA. Variance is scale-dependent, and PCA chooses directions of maximum variance. A feature measured in dollars (range: thousands) will dominate the variance calculation over one measured in years (range: tens), purely because of its scale, not because it's more informative. There are no right answers here, only trade-offs. With PCA the trade-off is interpretability. Component 1 is something like $0.31 \times \text{income} - 0.22 \times \text{age} + 0.18 \times \text{tenure} + \ldots$: a blend rather than a concept. Say a prediction needs to be explained in terms of a specific original variable, for a regulator, or a customer. Tree-based feature importance on the raw features is the better fit.

15.2.4 A Note on Kernel PCA

PCA assumes the informative structure in the data is linear: one direction, or a few, that variance stretches along. When it isn't, kernel PCA applies the same kernel trick Chapter 11 introduces for Support Vector Machines. It implicitly maps the data into a higher-dimensional space via a kernel function, then runs ordinary PCA there. Kernel PCA is a different tool, not a drop-in upgrade. The resulting components are harder to interpret. There's no simple way back to the original feature space, the way ordinary PCA has. The choice of kernel and its parameters becomes its own modelling decision. It's worth reaching for when a scatter plot, or a failed linear PCA, suggests the real structure curves through feature space rather than sitting along a straight line.

15.2.5 t-SNE and UMAP: Visualisation, Not Modelling Features

t-SNE and UMAP are non-linear dimensionality reduction techniques. They're almost always used to compress data to 2 or 3 dimensions for visualisation, rather than as inputs to a downstream model. Both preserve *local* neighbourhood structure, meaning points close together in the original high-dimensional space should stay close together in the embedding, rather than PCA's global objective of maximising total variance. This makes them better at revealing cluster structure visually. PCA can smear well-separated real clusters together if the axis of maximum variance doesn't align with what separates them. t-SNE or UMAP will usually recover more of that separation, though not always cleanly.

Listing 15.1 Three real clusters, PCA vs. UMAP vs. t-SNE, when a noisy nuisance dimension dominates variance

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.metrics import silhouette_score
import umap

# Three clusters, well separated in 2 dimensions, plus a
# third, high-variance dimension that carries no cluster
# signal at all -- pure noise, added on top.
X_clusters, y = make_blobs(
    n_samples=300, centers=3, n_features=2,
    cluster_std=0.5, center_box=(-2, 2), random_state=42,
)
rng = np.random.RandomState(42)
noise_dim = rng.normal(loc=0, scale=10, size=(300, 1))
```

```
X = np.hstack([X_clusters, noise_dim])

X_pca = PCA(n_components=2, random_state=42).fit_transform(X)
X_umap = umap.UMAP(n_components=2, random_state=42).fit_transform(X)
X_tsne = TSNE(n_components=2, random_state=42).fit_transform(X)

pca_sil = silhouette_score(X_pca, y)
umap_sil = silhouette_score(X_umap, y)
tsne_sil = silhouette_score(X_tsne, y)
print(f'PCA silhouette:  {pca_sil:.4f}')
print(f'UMAP silhouette: {umap_sil:.4f}')
print(f't-SNE silhouette (default perplexity=30): {tsne_sil:.4f}')

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis',
                alpha=0.75, s=30)
axes[0].set_title(f'PCA (silhouette = {pca_sil:.3f})')
axes[0].set_xlabel('PC1')
axes[0].set_ylabel('PC2')
axes[1].scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap='viridis',
                alpha=0.75, s=30)
axes[1].set_title(f'UMAP (silhouette = {umap_sil:.3f})')
axes[1].set_xlabel('UMAP1')
axes[1].set_ylabel('UMAP2')
plt.tight_layout()
plt.show()
```

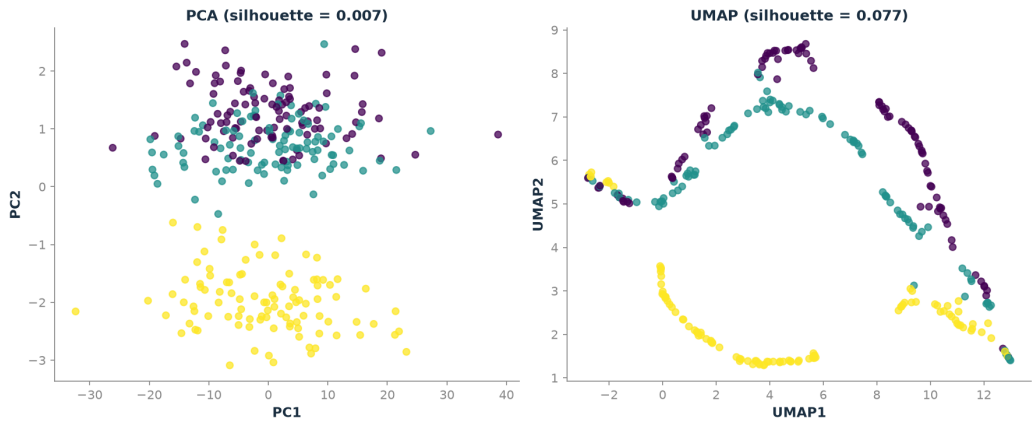


Figure 15.1: The same three clusters, embedded by PCA (left) and UMAP (right), with a single noisy nuisance dimension added to the original two.

Figure 15.1 is the honest picture, not the idealised one. PCA's top component chases the noisy nuisance dimension, since it has by far the largest variance. Two of the three clusters (the dark and mid-tone points) merge into a single smeared mass, while the third (the lightest) stays visibly apart. Silhouette score: 0.0073, close to the 0 that means "no cluster structure visible at all." UMAP does better, but "better" here means partial, not perfect. The lightest cluster separates cleanly, and the dark and mid-tone clusters pull apart along most of their length, though they still touch where the embedding curls back on itself. Silhouette score 0.0774, roughly ten times higher than PCA's: a real and measured improvement, not a clean sweep.

t-SNE, run on the same data with its default settings, is the surprise: 0.0033, worse than PCA, not better. That's not a bug in the method. t-SNE builds its neighbourhood graph from distances measured in the *original* feature space, the same space where the noise dimension's huge scale already dominates every pairwise distance. A method built to preserve local neighbourhoods still inherits whatever those neighbourhoods look like in the raw data, noise dimension included, unless something else, a smaller perplexity, a prior scaling step, counteracts it. Rerunning with `perplexity=12` instead of the default 30 gives 0.107, better than UMAP, on the exact same data. That swing, from worse-than-PCA to better-than-UMAP on one hyperparameter change, is the sensitivity this chapter's Trade-offs section warns about, made concrete.

The moral isn't "UMAP is better than PCA," and it isn't "t-SNE and UMAP always fix this" either. PCA's global-variance objective can be actively misled by a single noisy, high-variance column that has nothing to do with the structure that matters. A local-structure method recovers *some* of what PCA lost, but only if its own settings are tuned to the data, and a default configuration is no guarantee of that. That's exactly why it's worth looking at the real picture, with real numbers, instead of assuming the textbook one.

There are three separate reasons their output shouldn't feed a model, though. First: both are typically non-parametric in their classic form. There's no simple `transform()` for new data with the same guarantee as the fitted embedding. UMAP supports `transform()` with caveats, while classic t-SNE does not, without retraining on the combined dataset. Second: distances *between* clusters in the embedding aren't meaningful. Only local neighbourhood structure within a cluster is preserved reliably, so two clusters that look far apart aren't necessarily "more different" than two that look close. Third: both are stochastic and sensitive to hyperparameters (perplexity for t-SNE; `n_neighbors` and `min_dist` for UMAP). Rerunning with a different seed or parameter can meaningfully change the layout. That's fine for a one-off exploratory plot, but a poor foundation for a reproducible production feature. Use t-SNE and UMAP to look at data: check whether classes separate, spot outlier clusters, sanity-check a clustering result from Chapter 19. Use PCA, or regular feature selection, when reduced dimensions need to feed a model.

15.3 Trade-offs and Limits

Selection preserves interpretability and is safe to defend to a stakeholder ("we use income and tenure"). Extraction squeezes out more redundancy but gives up that interpretability entirely. Filters are cheap and model-agnostic, but ignore feature interactions. Wrappers are thorough, but expensive and specific to whichever model did the searching. Embedded methods are the best per-cost trade-off for most tabular problems, and are the reasonable place to start. Between PCA and t-SNE/UMAP: PCA is linear, deterministic, and approximately reversible. An approximation of the original data can be reconstructed from it, which makes it suitable both for visualisation and as a preprocessing step for a downstream model. t-SNE and UMAP are non-linear and much better for visual cluster discovery, but they're the wrong tool whenever reduced features need to be deployed and explained.

It's also worth asking whether this chapter's machinery is needed at all. Tree-based models (Random Forest, XGBoost, LightGBM) are relatively robust to irrelevant features. They won't split on features that don't help. Feature selection matters most for linear models, distance-based models (KNN, K-Means), and any case where training cost or interpretability is a real constraint. It is not a required step before training.

15.4 Procedures

Procedure 15.1 (Drop Low-Variance and Highly Correlated Features). *When to use:* as a first, cheap pass before any modelling, to remove features that are almost constant or redundant with another feature.

Listing 15.2 Dropping near-constant columns and one of each highly-correlated pair

```
from sklearn.feature_selection import VarianceThreshold
import pandas as pd
import numpy as np

# Drop near-constant columns (threshold is variance, not
# std, tune for your scale)
selector = VarianceThreshold(threshold=0.01)
X_reduced = selector.fit_transform(X_train)
kept_columns = X_train.columns[selector.get_support()]

# Drop one of each highly-correlated pair
corr_matrix = X_train.corr().abs()
upper_mask = np.triu(np.ones(corr_matrix.shape), k=1).astype(bool)
upper = corr_matrix.where(upper_mask)
to_drop = [col for col in upper.columns if any(upper[col] > 0.95)]
X_train_decorrelated = X_train.drop(columns=to_drop)
```

Common mistake: running `VarianceThreshold` on unscaled data. A feature ranging 0-1,000,000 will always look "high variance" even if it's useless, while a feature ranging 0-1 might get dropped even if it's highly predictive. Scale first, or reserve variance thresholding for features already on comparable scales such as one-hot encoded columns.

Procedure 15.2 (Select Features by Mutual Information). *When to use:* a fast, non-linear-aware ranking of features against the target, without training a full model.

Listing 15.3 Ranking and selecting top features by mutual information with the target

```
from functools import partial
from sklearn.feature_selection import (
    mutual_info_classif,
    mutual_info_regression,
    SelectKBest,
)

mi_scores = mutual_info_classif(X_train, y_train, random_state=42)
mi_ranking = (
    pd.Series(mi_scores, index=X_train.columns)
    .sort_values(ascending=False)
)

# mutual_info_classif is randomised: seed it here too
mi_seeded = partial(mutual_info_classif, random_state=42)
selector = SelectKBest(score_func=mi_seeded, k=10)
X_selected = selector.fit_transform(X_train, y_train)
selected_columns = X_train.columns[selector.get_support()]
# For a regression target, use mutual_info_regression instead
```

Common mistake: computing mutual information, or any selection statistic, on the full dataset before the train/test split: that leaks test-set information into the feature-selection decision. Fit the selector on the training fold only, then apply the same selected columns to the test set.

Procedure 15.3 (Run Recursive Feature Elimination). *When to use:* finding the feature subset that most helps a specific model, when retraining it repeatedly is affordable.

Listing 15.4 RFE with a fixed target count, and RFECV letting cross-validation choose the count

```
from sklearn.feature_selection import RFE, RFECV
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100, random_state=42)
rfe = RFE(estimator=model, n_features_to_select=10, step=1)
rfe.fit(X_train, y_train)
selected_columns = X_train.columns[rfe.support_]

# Let cross-validation choose the count automatically
# (preferred: avoids guessing)
rfecv = RFECV(estimator=model, step=1, cv=5, scoring='accuracy')
rfecv.fit(X_train, y_train)
print(f"Optimal number of features: {rfecv.n_features_}")
selected_columns = X_train.columns[rfecv.support_]
```

Common mistake: using a slow model (a deep neural network, a large XGBoost ensemble) inside RFE. Each elimination step retrains from scratch, so runtime multiplies by the number of features removed. Use a fast, well-regularised model for the search; it's not being deployed, only used to rank features.

Procedure 15.4 (Use Tree-Based Feature Importance for Selection). *When to use:* a tree ensemble is already in use, or affordable to fit, and selection is wanted essentially for free.

Listing 15.5 Selecting features above the median Random Forest importance

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel

rf = RandomForestClassifier(n_estimators=200, random_state=42)
rf.fit(X_train, y_train)

importances = (
    pd.Series(rf.feature_importances_, index=X_train.columns)
    .sort_values(ascending=False)
)

selector = SelectFromModel(rf, prefit=True, threshold='median')
X_selected = selector.transform(X_train)
```

Common mistake: trusting importance rankings blindly when features are highly correlated. Tree models split arbitrarily between two near-identical correlated features, diluting importance across both and making each look less important than a single, non-redundant version would. Decorrelate first (Procedure 15.1) if this matters.

Procedure 15.5 (Choose the Number of PCA Components). *When to use:* after deciding PCA is appropriate (linear structure, a need for a reproducible transform) before fitting the final model.

Listing 15.6 Scree plot and cumulative variance to choose a PCA component count, or letting sklearn pick one directly

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np

# Always scale first
X_scaled = StandardScaler().fit_transform(X_train)

pca_full = PCA().fit(X_scaled)
cumulative_variance = np.cumsum(pca_full.explained_variance_ratio_)

plt.plot(
    range(1, len(cumulative_variance) + 1),
    cumulative_variance,
    marker='o',
)
plt.axhline(
    0.95, color='red', linestyle='--', label='95% variance'
)
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance')
plt.legend()
plt.show()

# Or just ask for a variance target directly:
# sklearn picks the count
pca = PCA(n_components=0.95)
X_pca = pca.fit_transform(X_scaled)
print(f"Components needed for 95% variance: {pca.n_components_}")
```

Common mistake: fitting `PCA` on training and test data combined "to get a stable transform": this leaks test-set variance structure into the transform. Fit on training data only, and call `.transform()`, never `.fit_transform()`, on the test set.

Procedure 15.6 (Interpret PCA Loadings). *When to use:* explaining, even approximately, what a principal component "means."

Listing 15.7 Reading which original features drive a given principal component

```
pca = PCA(n_components=3).fit(X_scaled)

loadings = pd.DataFrame(
    pca.components_.T,
    columns=[f'PC{i+1}' for i in range(3)],
    index=X_train.columns,
)

print(loadings['PC1'].abs().sort_values(ascending=False).head(5))
```

A large positive or negative loading means that feature strongly drives the component: the sign indicates direction only, not importance. If PC1 loads heavily and positively on `income`, `credit_limit`, and `spend`, it's roughly "overall financial scale," but that's a rough label, not a rigorous claim.

15.5 Worked Example

The tutorial notebook builds a synthetic classification dataset with 50 features: 10 truly informative, 10 redundant (linear combinations of the informative ones), and 30 pure noise. It then runs every selection and extraction technique against it and compares downstream Random Forest accuracy and fit time. A baseline using all 50 features scores:

Listing 15.8 Baseline evaluation function used throughout the comparison

```
def evaluate(X_tr, X_te, y_tr, y_te, label, model=None):
    if model is None:
        model = RandomForestClassifier(
            n_estimators=200, random_state=RANDOM_SEED
        )
    start = time.time()
    model.fit(X_tr, y_tr)
    fit_time = time.time() - start
```

```
    acc = model.score(X_te, y_te)
    print(
        f"{label:32s} | n_features={X_tr.shape[1]:3d} | "
        f"fit_time={fit_time:.3f}s | test_accuracy={acc:.4f}"
    )
    return {
        "label": label,
        "n_features": X_tr.shape[1],
        "fit_time": fit_time,
        "accuracy": acc,
    }
```

Running the full comparison on a 1,500-row training split (2,000 samples, 25% held out, `random_state=42` throughout) produces:

Method	Features	Fit time (s)	Test accuracy
Wrapper: RFECV	26	0.552	0.928
Embedded: RF importance (median threshold)	25	0.550	0.922
Baseline (all 50 features)	50	0.754	0.912
Filter: mutual information (top 15)	15	0.397	0.884
Extraction: PCA (37 components)	37	0.705	0.868

The pattern matches what this chapter's Trade-offs section predicts. RFECV and the embedded Random Forest importance method don't just match the 50-feature baseline, they beat it, using roughly half the columns and shedding noise features that were otherwise available for the model to overfit on. The mutual-information filter and PCA extraction trail the baseline here, by 2.8 and 4.4 points. PCA's 37-component result is a reminder. Sometimes real informative structure is spread across many original dimensions, not just redundancy to compress. Then extraction can lose a little relative to a wrapper method that gets to see actual downstream accuracy during selection. Mutual information's top-15 list: `feature_0`, `feature_5`, `feature_6`, `feature_7`, `feature_9`, `feature_18`, `feature_23`, `feature_25`, `feature_26`, `feature_37`, `feature_38`, `feature_43`, `feature_45`, `feature_48`, `feature_49`. Those names don't say which features are informative: `make_classification` shuffles its columns by default. The notebook rebuilds the data with `shuffle=False`, which puts the informative and redundant features in `feature_0` to `feature_19`, to check the filter against the ground truth.

On the extraction side, fit `PCA()` with no component limit on the scaled 50-feature training set and read the cumulative explained-variance curve. **37 components are needed to reach 95% of the variance.** Not a dramatic compression. After scaling, every column

has the same variance, so the 30 pure-noise columns hold about 60% of the total. PCA ranks directions by variance alone, so it can't skip the noise, and it needs most of those noise directions to reach 95%.

Listing 15.9 Finding the component count needed for 95% cumulative explained variance

```
pca_full = PCA(random_state=RANDOM_SEED).fit(X_train_scaled)
cumulative_variance = np.cumsum(pca_full.explained_variance_ratio_)
n_components_95 = np.argmax(cumulative_variance >= 0.95) + 1
print(
    f"Components needed for 95% variance: {n_components_95} "
    "(out of 50 original features)"
)
```

The second half of the notebook switches from modelling to pure visualisation. It loads the classic 64-pixel handwritten-digits dataset (1,797 samples, 10 classes) and compares PCA, t-SNE, and UMAP as they each compress it to 2D. Timing the three on this machine: PCA runs in 0.05s, t-SNE in 1.54s, and UMAP in 8.58s. PCA's near-instant linear projection is the expected result. UMAP's time here is mostly one-off compilation on its first call. On larger datasets, UMAP is usually faster than t-SNE. Visually, plotting each embedding coloured by true digit class, PCA produces a reasonable but often overlapping scatter, since it optimises for global variance rather than class separation. t-SNE and UMAP never see the labels, yet typically produce much tighter, more visually distinct clusters per digit. Both are explicitly preserving local neighbourhood structure rather than total spread.

15.6 AWS SageMaker Notes

AWS SageMaker Equivalent

```
# SageMaker's built-in PCA algorithm: useful for very
# large datasets that don't fit in memory on a single
# machine (distributed PCA)
import sagemaker
from sagemaker import PCA as SageMakerPCA

pca_estimator = SageMakerPCA(
    role=role,
    instance_count=1,
    instance_type='ml.m5.xlarge',
    num_components=10,
    mini_batch_size=200,
)
# pca_estimator.fit(train_data_s3_uri)
```

```
# For feature selection at scale, SageMaker Data Wrangler
# includes a "Balance and Feature Correlation" analysis
# plus a "Feature Importance" report that combines with
# the same drop-column transforms used above.
```

When to prefer SageMaker's built-in PCA: only when the dataset doesn't fit in memory on one machine. For anything that fits comfortably in a pandas DataFrame, `sklearn.decomposition.PCA` is simpler and just as correct.

15.7 History and Further Reading

Murphy's Introduction, Chapter 20, works through the PCA derivation both via eigendecomposition and via SVD with worked examples (Murphy 2022). Burkov gives a concise treatment of dimensionality reduction and feature selection side by side (Burkov 2019, 9). Géron's *Hands-On Machine Learning*, Ch. 8, works through PCA and kernel PCA with Scikit-Learn code, a practical complement to this chapter's derivations and its brief kernel PCA note above. It also covers LLE, a dimensionality-reduction technique not included here (Géron 2019). Wattenberg et al.'s "How to Use t-SNE Effectively" (distill.pub, 2016) is essential reading before trusting any t-SNE plot. The UMAP documentation's "How UMAP Works" gives an accessible account of the algorithm's topological motivation. Both are worth reading in full before either technique is used to justify a claim about data structure.

15.8 How This Connects

This chapter builds on Chapter 3's linear algebra (PCA is an eigendecomposition of the covariance matrix) and on the mutual information machinery from Chapter 5, one of the filter-selection criteria used above. It also assumes Chapter 14's feature engineering has already produced the candidate columns being pruned here. It enables Chapter 16 (Model Evaluation), since a smaller, cleaner feature set is usually easier to evaluate and less prone to overfitting-driven metric inflation. It also enables Chapter 19 (Unsupervised Learning), where t-SNE/UMAP visualisation is a standard way to sanity-check a clustering result. It's often confused with feature engineering (Chapter 14): engineering *creates* candidate features, while this chapter *reduces* them.

15.9 Exercises

1. **[W]** Explain in your own words why feature selection preserves interpretability while feature extraction does not, using the "income is still income" framing from this chapter.

2. [**W**] Classify each of the following as a filter, wrapper, or embedded method, and give one reason: (a) dropping features with variance below 0.01; (b) RFECV; (c) keeping features with Random Forest importance above the median.

3. [**A**] Using Listing 15.3 as a starting point, rerun mutual-information selection with `k=5` instead of `k=10` on the synthetic dataset from Listing 15.8, and report whether the resulting 5 features are a subset of the 15 reported in the Worked Example.

4. [**A**] Using Listing 15.9's logic, find how many components are needed to reach 99% cumulative explained variance on the same 50-feature training set, and compare the result to the 37 needed for 95%.

5. [**A**] Reproduce the digits-dataset PCA vs. t-SNE timing comparison from the Worked Example, but with `perplexity=5` instead of `perplexity=30` in the t-SNE call. Describe qualitatively how the resulting 2D plot's cluster tightness changes.

6. [**E**] The Worked Example shows RFECV and embedded Random Forest importance both *beating* the 50-feature baseline. Using the curse of dimensionality from this chapter's Exposition, explain why removing features can improve accuracy rather than merely preserving it.

7. [**E**] A regulator requires that a credit-scoring model's predictions be explainable in terms of the original applicant-provided variables. A colleague proposes using PCA to cut a 200-feature dataset down to 20 components first. Write two or three sentences explaining why this is likely to cause a problem, and what you'd suggest instead.

8. [**R**] Read the Wattenberg et al. "How to Use t-SNE Effectively" piece cited above. Identify one t-SNE pitfall it describes that this chapter did not mention, and explain how it would change the interpretation of the digits-dataset t-SNE plot in the Worked Example.

15.10 Mental Model

Feature engineering fills a toolbox. Feature selection and dimensionality reduction decide which tools are needed for the job in front of you. Selection removes tools that won't be used. Extraction welds several overlapping tools down into one multi-purpose one. Both exist because a toolbox that's overflowing slows everything down, hurting more than it helps.

16 Model Evaluation

Once a model is trained, a simple question follows: is it good? The naive answer ("measure accuracy") is almost the wrong one. Choosing the wrong metric means optimising for the wrong thing and shipping a model that fails in production in ways that are surprising to whoever picked the metric. This chapter is about learning to ask "good for what?" before measuring anything.

16.1 Motivation

Every evaluation metric is a choice about what to care about. A fraud-detection model that labels every transaction "not fraud" scores roughly 99.9% accuracy on a typical fraud dataset, because fraud is rare. That model is useless. It catches zero fraud while looking nearly perfect. The metric didn't lie. Well, not exactly. It answered a question nobody asked. The foundation of all classification evaluation is the confusion matrix: a table that counts what a model got right and wrong, broken down by type. Every metric that follows is a formula built on top of that matrix.

16.2 Exposition

16.2.1 The Confusion Matrix

	Predicted Positive	Predicted Negative
Actual positive	True Positive (TP)	False Negative (FN)
Actual negative	False Positive (FP)	True Negative (TN)

A false positive is a "false alarm" (Type I error). A false negative is a "missed it" (Type II error). Which one costs more depends entirely on domain. In cancer screening, missing a cancer (FN) is catastrophic. In spam filtering, flagging a legitimate email (FP) is merely annoying. No metric decision makes sense before that cost asymmetry is written down and understood.

16.2.2 Classification Metrics

Accuracy $((TP + TN)/(TP + TN + FP + FN))$ is useful only when classes are balanced and both error types cost the same. That combination is rare in practice. **Precision** $(TP/(TP + FP))$ answers "of everything predicted positive, how many were." It matters when false positives are costly. **Recall** $(TP/(TP + FN))$ answers "of everything actually positive, how many were caught." It matters when false negatives are costly. **F1** (the harmonic mean of precision and recall) gives a single balanced number when both matter and classes are imbalanced. **ROC-AUC** (Area Under the Receiver Operating Characteristic Curve) measures ranking quality across every possible decision threshold at once (1.0 perfect, 0.5 random). It's useful for comparing models before a threshold is chosen. **PR-AUC** (Area Under the Precision-Recall Curve) does the same, but focuses on the positive class specifically. It's the more honest choice under heavy imbalance, where ROC-AUC can look deceptively good. **Log loss** penalises confident wrong predictions heavily. It matters when the actual probability, not just the ranking, is going to drive a decision.

> **i** Why Harmonic, Not Arithmetic?
>
> The familiar mean, $(a + b)/2$, treats a strong score and a weak one as interchangeable. Precision 1.0 and recall 0.01 average out to 0.505, which reads as "roughly middling" for a model that in practice catches almost nothing. The harmonic mean, $2ab/(a + b)$, doesn't average the two numbers so much as average their *reciprocals*. That pulls the result sharply towards whichever number is smaller. The same precision 1.0 and recall 0.01 give a harmonic mean of 0.0198, close to the useless 0.01 and nothing like the reassuring 0.505 the arithmetic mean reported. That's exactly the property F1 needs. A model has to do reasonably well on *both* precision and recall to get a good F1 score. A harmonic mean can't be dragged up by one strong number covering for one weak one the way an arithmetic mean can.

16.2.3 The Precision-Recall Trade-off

A classifier outputs a probability, not a label. A threshold (commonly 0.5) turns that probability into a decision. Lowering the threshold predicts positive more often, raising recall at the cost of precision. Raising the threshold does the opposite. There is no free lunch. The PR curve shows every trade-off point at once. The business context, what an FP costs versus what an FN costs, is what tells you where on that curve to operate.

16.2.4 Regression Metrics

MSE $(\text{mean}((y - \hat{y})^2))$ penalises large errors quadratically and is differentiable, which is why it's used during training. RMSE $(\sqrt{\text{MSE}})$ is the same information in the target's own units.

MAE (mean($|y - \hat{y}|$)) is less sensitive to outliers than RMSE. A large RMSE/MAE ratio, rule of thumb: above roughly 2x, is a diagnostic that outliers are present and influencing RMSE disproportionately. MAPE gives scale-invariant percentage error but breaks when the target can be zero. R^2 gives the proportion of variance: 1 is perfect, 0 means no better than predicting the mean, and negative means actively worse than that trivial baseline.

16.2.5 Calibration

Calibration asks a simple question: when a model says 70% probability, is it right 70% of the time? A model can separate positives from negatives well, good discrimination, e.g. high ROC-AUC, while still being poorly calibrated. Its probabilities are systematically over- or under-confident. Calibration matters when the probability *itself* drives a decision. Insurance pricing, medical risk scores, and financial models all need calibrated probabilities, not just correctly-ordered ones. The reliability diagram plots predicted-probability bins against observed frequency. A perfectly calibrated model lies on the diagonal. Random Forests are a common source of S-shaped miscalibration. Their probabilities rarely reach 0 or 1, so they are under-confident at the extremes. Naive Bayes shows the opposite pattern: over-confident. Fixable with isotonic regression or Platt scaling via `CalibratedClassifierCV`.

16.2.6 The Business Metric Gap

An ML metric and a business metric are different things. Minimising RMSE on a demand forecast doesn't maximise profit, because the costs of over-stocking and under-stocking are rarely symmetric. The right question is "what does a unit improvement in this ML metric get us in business terms." Not "is this ML metric improving."

16.3 Trade-offs and Limits

Accuracy is acceptable with balanced classes, symmetric error costs, and a rough model-to-model comparison in mind. It misleads under class imbalance: prefer F1 or PR-AUC there. It misleads when error costs differ: choose precision or recall based on which error is expensive. And it misleads whenever probabilistic output itself matters: prefer log loss plus calibration checking there instead.

16.4 Procedures

Procedure 16.1 (Compute All Classification Metrics at Once). *When to use:* immediately after getting predictions from any classifier.

Listing 16.1 Full metric suite from one set of predictions

```
from sklearn.metrics import classification_report, confusion_matrix

y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]

print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
```

Procedure 16.2 (Plot the ROC and Precision-Recall Curves). *When to use:* comparing models' discrimination ability, or picking a threshold.

Listing 16.2 ROC and PR curves from predicted probabilities

```
import matplotlib.pyplot as plt
from sklearn.metrics import (
    roc_curve,
    roc_auc_score,
    precision_recall_curve,
    average_precision_score,
)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

fpr, tpr, _ = roc_curve(y_test, y_proba)
auc = roc_auc_score(y_test, y_proba)
axes[0].plot(fpr, tpr, color='#4E79A7',
             label=f'Logistic Regression (AUC = {auc:.3f})')
axes[0].plot(
    [0, 1], [0, 1], color='#999999', linestyle='--', linewidth=1.5,
    label='Random classifier (AUC = 0.5)',
)
axes[0].set_xlabel('False Positive Rate')
axes[0].set_ylabel('True Positive Rate (Recall)')
axes[0].set_title('ROC Curve')
axes[0].legend()

precision, recall, _ = precision_recall_curve(y_test, y_proba)
ap = average_precision_score(y_test, y_proba)
axes[1].plot(recall, precision, color='#E15759',
             label=f'Logistic Regression (AP = {ap:.3f})')
axes[1].axhline(
```

```
    y_test.mean(), color='#999999', linestyle='--', linewidth=1.5,
    label=f'No-skill baseline ({y_test.mean():.2f})',
)
axes[1].set_xlabel('Recall')
axes[1].set_ylabel('Precision')
axes[1].set_title('Precision-Recall Curve')
axes[1].legend()

plt.tight_layout()
plt.show()
```

Procedure 16.3 (Adjust the Decision Threshold). *When to use:* when FP and FN costs differ, or the default 0.5 threshold isn't optimal.

Listing 16.3 Sweeping thresholds to maximise F1

```
import numpy as np
from sklearn.metrics import precision_recall_curve

precision, recall, thresholds = precision_recall_curve(
    y_test, y_proba
)
f1_scores = 2 * precision * recall / (precision + recall + 1e-8)
best_threshold = thresholds[np.argmax(f1_scores[:-1])]
```

Procedure 16.4 (Compute and Fix a Calibration Curve). *When to use:* when predicted probabilities drive a decision directly.

Listing 16.4 Diagnosing and fixing calibration

```
from sklearn.calibration import (
    calibration_curve,
    CalibratedClassifierCV,
)

prob_true, prob_pred = calibration_curve(
    y_test, y_proba, n_bins=10
)

calibrated = CalibratedClassifierCV(
    model, method='isotonic', cv=5
)
calibrated.fit(X_train, y_train)
```

Common mistake: trusting a raw model's `predict_proba` output as a real probability without checking calibration first. Random Forests, for example, are often under-confident at the extremes.

16.5 Worked Example

On the Breast Cancer Wisconsin dataset (569 examples; logistic regression on standardised features, 80/20 stratified split), the confusion matrix on the 114-example test set comes out as:

Listing 16.5 Real confusion matrix and derived metrics

```
cm = confusion_matrix(y_test, y_pred)
# [[41  1]
#  [ 1 71]]
```

TN=41, FP=1, FN=1, TP=71. That gives accuracy 0.9825, precision 0.9861, recall 0.9861, F1 0.9861, ROC-AUC 0.9954, PR-AUC 0.9971, and log loss 0.0777. On this dataset, roughly balanced (37%/63%) and well-separated by the available features, accuracy and F1 tell essentially the same story. That's the signal that accuracy is *acceptable* here specifically because the imbalance is mild and both error types are similarly rare. Running Procedure 16.2 against this same model and test set produces Figure 16.1.

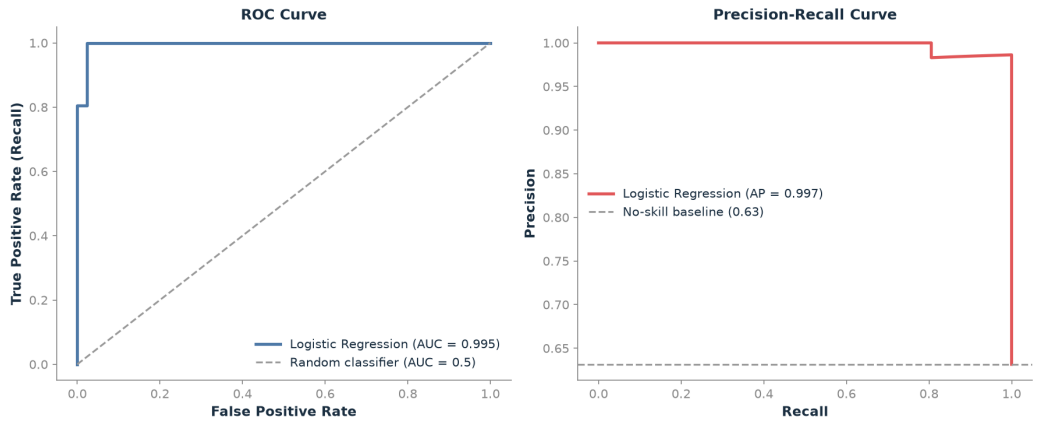


Figure 16.1: ROC and precision-recall curves for the logistic regression on the breast-cancer test set. The ROC curve's AUC (0.995) and the PR curve's average precision (0.997) both confirm near-perfect ranking; the PR curve's steep drop at the very end is the one false positive and one false negative this model makes.

Both curves stay close to their perfect corner for almost their entire length. The ROC curve hugs the top-left. The PR curve stays near precision 1.0 until the last handful of points, where the model's two mistakes show up as a sharp drop rather than a gradual slide. That's the visual signature of a model that is right almost everywhere and wrong on only a couple of hard cases.

Sweeping the decision threshold to maximise F1, rather than using the default 0.5, finds a best threshold of 0.366. F1 rises slightly, from 0.9861 to 0.9931: a small but real improvement, and a reminder that even on an easy, well-separated dataset, the default threshold isn't automatically optimal.

Contrast this with a hypothetical, deliberately harder case: a fraud dataset where only 0.1% of transactions are fraudulent. A model that predicts "not fraud" every time scores 0.999 accuracy, literally 99.9% correct, while catching exactly zero fraud (recall = 0, precision undefined since $TP = 0$). This isn't a contrived edge case. It's the direct arithmetic consequence of accuracy rewarding the majority class, regardless of what the minority class needed the model to do. The breast-cancer dataset's near-balanced classes are exactly why accuracy behaved reasonably there, and exactly why the same metric would be a lie on a heavily imbalanced problem.

AWS SageMaker Equivalent

```
from sagemaker.model_monitor import ModelQualityMonitor

monitor = ModelQualityMonitor(
    role=role, instance_count=1, instance_type="ml.m5.xlarge"
)
monitor.suggest_baseline(
    baseline_dataset=(
        "s3://my-bucket/validation-with-predictions.csv"
    ),
    dataset_format=DatasetFormat.csv(header=True),
    output_s3_uri="s3://my-bucket/baseline/",
    problem_type="BinaryClassification",
    inference_attribute="prediction",
    probability_attribute="probability",
)
monitor.create_monitoring_schedule(
    endpoint_input=endpoint_name,
    schedule_cron_expression="cron(0 * ? * * *)",
)
```

When to prefer SageMaker Model Monitor: when the metric needs to be tracked continuously against live production traffic, not just computed once on a static test set. It flags metric degradation (e.g. a drop in precision) automatically as the input distribution drifts, rather than waiting for someone to notice.

16.6 History and Further Reading

Thakur, Ch. 6, gives practical metric selection with code examples (Thakur 2020). Géron's *Hands-On Machine Learning*, Ch. 3, works through the confusion matrix, precision, recall, and ROC curves on a single running example, a good practical companion to this chapter's metrics (Géron 2019). Murphy's Introduction, Ch. 5, gives the formal statistical treatment of evaluation (Murphy 2022). Davis and Goadrich's paper on the relationship between precision-recall and ROC curves explains why PR-AUC matters specifically under imbalance (Davis and Goadrich 2006).

16.7 How This Connects

This chapter builds on Chapter 11 (a model is needed before it can be evaluated) and on Chapter 13's cross-validation, since proper evaluation requires the right split strategy applied consistently. It's often confused with the loss function used during training. The loss is what's minimised while fitting (often cross-entropy). The evaluation metric is what's measured afterwards (often F1 or a business-specific number). They are usually different things, measuring related but distinct properties of the same model.

16.8 Exercises

1. **[W]** Using the confusion matrix definitions, explain in one sentence each what a false positive and a false negative mean for a spam filter, and state which one is more costly there.
2. **[W]** State which metric (precision, recall, or F1) is most appropriate for: (a) a disease-screening test; (b) a spam filter; (c) a general-purpose classifier comparison with no stated cost asymmetry.
3. **[A]** Using Listing 16.5, compute precision and recall by hand from TN=41, FP=1, FN=1, TP=71, and confirm they match the reported 0.9861 figures.
4. **[A]** Using this chapter's hypothetical 0.1%-fraud scenario, compute the accuracy of a model that predicts "fraud" for every transaction instead (the opposite extreme), and state its precision and recall.
5. **[E]** A customer reports "our fraud model has 99.9% accuracy" as evidence it's ready to ship. Using this chapter's Worked Example, write the two or three sentences you'd say before agreeing.

6. **[E]** A Random Forest's `predict_proba` output is being used directly to set insurance premiums. Using this chapter's Calibration section, explain what could go wrong and what you'd check before trusting those numbers.
7. **[R]** Read the Davis and Goadrich (2006) paper's abstract. Identify the specific claim it makes about ROC curves under class imbalance, and explain how it relates to this chapter's PR-AUC recommendation for rare-event detection.

16.9 Mental Model

The confusion matrix is the ground truth. Every metric is just a different question asked of that table. Before choosing one, write down what a false positive and a false negative cost in the specific business context. The right metric usually becomes obvious once those costs are made visible.

17 Hyperparameter Optimisation

Every machine learning algorithm has knobs, levers, and dials that must be set before training starts. How many trees. How deep. How fast the model should learn. These hyperparameters aren't learned from data. They're chosen. Choose poorly and a potentially excellent algorithm underperforms. Choose wisely and a merely adequate algorithm will beat a badly-tuned state-of-the-art one. This chapter is about searching that space systematically, since there's no derivative of "validation accuracy with respect to number of trees" to follow.

17.1 Motivation

Hyperparameter optimisation is a search problem. There's a function that can be evaluated (train a model, measure validation performance) but not differentiated, in a space that's high-dimensional and expensive to explore. Each candidate configuration means training and validating a full model. In practice, random search beats grid search for the same compute budget. That's counterintuitive until you understand the reason: hello again, dimensionality. In high-dimensional spaces most hyperparameters barely matter and a few dominate, so grid search wastes trials on exhaustive coverage of irrelevant dimensions. Random search samples the important ones more densely by not being tied to a fixed grid at all.

17.2 Exposition

17.2.1 Parameters vs. Hyperparameters

Parameters (model weights, tree split thresholds) are learned during training via optimisation. Hyperparameters (learning rate, `n_estimators`, `max_depth`) are set before training begins and aren't differentiable with respect to the training process. They have to be searched, not solved for.

17.2.2 Search Strategies

Grid search enumerates every combination in a defined grid. Five values across four hyperparameters is $5^4 = 625$ training runs. Adding a fifth hyperparameter with five values pushes that to 3,125. This is the curse of dimensionality applied to hyperparameter search. It's only practical with three or fewer hyperparameters and small grids.

Random search (Bergstra and Bengio 2012) samples random combinations from the search space instead of iterating through a fixed grid. With the same budget as grid search, it covers more of the dimensions that matter. If most hyperparameters are unimportant, then grid search wastes most of its budget holding the few important ones fixed. Random search doesn't have that structural weakness. That paper is one of the most influential practical ML papers of the last fifteen years. It's about 25 pages, and dense. For a Solutions Architect, understanding the impact is much more important than reading the paper.

Bayesian optimisation builds a probabilistic surrogate model of the objective function. That surrogate is a cheap stand-in for "validation score as a function of hyperparameters," fast to query, unlike the real thing: training and validating an actual model. It uses that surrogate to decide where to sample next. It trades off exploitation (sample near the best known point) against exploration (sample where uncertainty is high). Two surrogate choices are common. A Gaussian Process models the objective directly, and grows more confident near points already tried. A Tree-structured Parzen Estimator (TPE) instead models which hyperparameter values tend to produce good results versus bad ones. TPE is the default surrogate used by **Optuna**, the hyperparameter-search library this chapter's procedures build on. The surrogate updates after every evaluation. This is more efficient than random search when each evaluation is expensive. Optuna also gives automatic pruning (stopping bad trials early using intermediate results). Pruning is orthogonal to the search strategy. It can be layered onto random search or Bayesian optimisation.

17.2.3 The Validation Set Is Sacred

During hyperparameter search, the validation set scores every candidate configuration and becomes part of the optimisation process. That means the reported validation score after tuning is biased upward. Many configurations have been tried and the best-scoring one on that validation set was chosen. This is a selection process, not a neutral estimate. A held-out test set that never touches any part of the search is required to get an honest final estimate of performance.

17.3 Trade-offs and Limits

Strategy	Pros	Cons	Use when
Grid Search	Exhaustive over the grid, reproducible	Exponential in hyperparameter count	≤ 3 hyperparameters, fast model
Random Search	Simple, parallelisable, surprisingly effective	No guidance from past trials	Most situations, good default
Bayesian (Optuna)	Most sample-efficient, prunes bad trials	Trials run one after another, so less parallelism	Expensive-to-evaluate models

17.4 Procedures

Procedure 17.1 (Define a Search Space). *When to use:* before running any search strategy.

Listing 17.1 Grid, distribution, and Optuna search-space definitions

```
# Grid: discrete lists
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
}

# Random: distributions
from scipy.stats import loguniform, randint
param_distributions = {
    'n_estimators': randint(50, 500),
    'learning_rate': loguniform(0.01, 0.3),
}

# Optuna: suggest_* calls inside the objective function
def objective(trial):
    n_estimators = trial.suggest_int('n_estimators', 50, 500)
    lr = trial.suggest_float('learning_rate', 0.01, 0.3, log=True)
    # ... train and return the metric
```

Procedure 17.2 (Run `GridSearchCV`). *When to use:* three or fewer hyperparameters, fast model, small grid.

Listing 17.2 Exhaustive grid search

```
from sklearn.model_selection import GridSearchCV

gs = GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1,
)
gs.fit(X_train, y_train)
best_model = gs.best_estimator_
```

Procedure 17.3 (Run `RandomizedSearchCV`). *When to use:* the default for most situations: same API, wider coverage for the same budget.

Listing 17.3 Random search over distributions

```
from sklearn.model_selection import RandomizedSearchCV

rs = RandomizedSearchCV(
    model,
    param_distributions,
    n_iter=50,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1,
    random_state=42,
)
rs.fit(X_train, y_train)
```

Procedure 17.4 (Set Up an Optuna Study). *When to use:* expensive models where each evaluation costs real time.

Listing 17.4 TPE-based Bayesian search with Optuna

```
import optuna
optuna.logging.set_verbosity(optuna.logging.WARNING)

study = optuna.create_study(
    direction='maximize',
    sampler=optuna.samplers.TPESampler(seed=42),
)
study.optimize(objective, n_trials=100)

print(study.best_params, study.best_value)
```

Common mistake: evaluating the final model on the same validation folds used during the search and reporting that number as "the" performance. Always hold out an untouched test set and evaluate on it exactly once, after the search concludes.

17.5 Worked Example

Running grid search, random search, and Optuna's TPE sampler head-to-head on a synthetic, harder binary classification problem: 3,000 samples, 40 features, with label noise and overlapping classes, built so overfitting is a real risk. All three tune the same six XGBoost hyperparameters: `n_estimators`, `max_depth`, `learning_rate`, `subsample`, `colsample_bytree`, and `min_child_weight`. Grid search only has room for two values at the edge of each range. That's $2^6 = 64$ corner combinations. Random search and Optuna get the same 64-evaluation budget. But they sample continuously across the same six ranges, including the interior grid search can never reach:

Listing 17.5 Grid, random, and Optuna search under a matched 64-evaluation budget across six hyperparameters

```
param_grid = {
    'n_estimators': [100, 1000], 'max_depth': [2, 12],
    'learning_rate': [0.01, 0.3], 'subsample': [0.5, 1.0],
    'colsample_bytree': [0.5, 1.0], 'min_child_weight': [1, 10],
}  # two values per dimension: 2**6 = 64 corner combinations
gs = GridSearchCV(model, param_grid, cv=5, scoring='roc_auc', n_jobs=-1)

# Same ranges as the grid. Unlike Optuna below, this samples
# learning_rate uniformly, not on a log scale.
```

```
param_dist = {
    'n_estimators': randint(100, 1001), 'max_depth': randint(2, 13),
    'learning_rate': uniform(0.01, 0.29), 'subsample': uniform(0.5, 0.5),
    'colsample_bytree': uniform(0.5, 0.5), 'min_child_weight': randint(1, 11),
}
rs = RandomizedSearchCV(
    model, param_dist, n_iter=64, cv=5, scoring='roc_auc',
    random_state=42, n_jobs=-1,
)

# objective samples the same six ranges (learning_rate on a log
# scale), one trial at a time, with its 5 CV folds in parallel
study.optimize(objective, n_trials=64)
```

Measured results:

Strategy	Best CV AUC	Test AUC	Wall time
Grid search	0.9185	0.8933	55.5s
Random search	0.9198	0.8942	18.5s
Optuna (TPE)	**0.9225**	**0.8949**	171.5s

This time the measurement matches the theory it's testing. Random search beats grid search on both CV AUC (0.9198 vs. 0.9185) and test AUC (0.8942 vs. 0.8933). It's a modest edge, but a consistent one. It comes from a real structural weakness in the grid. With only two values per dimension, every one of grid search's 64 combinations sits at a corner of the search space: max or min `subsample`, max or min `max_depth`, and so on, in every combination. It can never place a trial in the interior, where the best setting for a hyperparameter often sits. Random search and Optuna sample continuously. They can reach the interior. Optuna's TPE sampler does best of the three on both metrics: CV AUC 0.9225, test AUC 0.8949. That's what the Trade-offs table predicts for the one strategy that uses past trials to guide the next one, instead of sampling blind.

Optuna's full 64-trial run also took the longest: 171.5 seconds, against random search's 18.5 and grid's 55.5. Judged on that number alone, Optuna looks like the expensive choice. Most of that gap is parallelism, not Optuna itself. Grid and random search spread all 320 fits across every core at once. Optuna runs one trial at a time, five folds in parallel, because each trial depends on the results before it. TPE's own overhead is well under a second. But its real advantage isn't wall time at a matched budget. It's needing fewer evaluations to reach a given quality. Tracked trial by trial, Optuna reached random search's final CV AUC (0.9198) after just 5 trials, in 5.7 seconds. That's faster than random search's full run, and almost ten times faster than grid's. This is the sample efficiency the Trade-offs table

promises for expensive-to-evaluate models. Stop Optuna once it's caught up, and it wins on accuracy and time both. Run it to a budget sized for a cruder strategy instead, and its sequential trials cost time with nothing to show for it.

AWS SageMaker Equivalent

```
from sagemaker.tuner import (
    HyperparameterTuner,
    IntegerParameter,
    ContinuousParameter,
)

tuner = HyperparameterTuner(
    estimator=estimator,
    objective_metric_name='validation:auc',
    hyperparameter_ranges={
        'n_estimators': IntegerParameter(50, 500),
        'max_depth': IntegerParameter(2, 10),
        'learning_rate': ContinuousParameter(0.01, 0.3),
    },
    max_jobs=50, max_parallel_jobs=5, strategy='Bayesian',
)
tuner.fit({'train': train_data_s3})
```

When to prefer SageMaker Hyperparameter Tuning: when the model takes hours to train and the search needs to parallelise across managed instances. For models that train in seconds to minutes (like this chapter's Worked Example), Optuna running locally is faster and cheaper than provisioning a fleet of training jobs.

17.6 History and Further Reading

Bergstra and Bengio's "Random Search for Hyper-Parameter Optimization" is the paper that established random search as the strong practical default (Bergstra and Bengio 2012). Thakur, Ch. 7, covers practical hyperparameter tuning with code (Thakur 2020). Géron's *Hands-On Machine Learning*, Ch. 2, shows GridSearchCV and RandomizedSearchCV directly on a full pipeline. It shows the mechanics behind the random-search argument above (Géron 2019). Murphy's Introduction, Ch. 5.3, gives the Bayesian optimisation formalism (Murphy 2022).

17.7 How This Connects

This chapter builds on Chapter 13's cross-validation: proper validation is what scores each candidate. It also builds on Chapter 16's model evaluation: the metric being optimised during search matters as much as the search strategy. It enables the ensemble hyperparameters from Chapter 12 (`n_estimators`, `max_depth`) to be tuned rather than guessed. It's often confused with feature selection: hyperparameter search asks what settings make an algorithm perform best. Feature selection asks what inputs make it perform best. Different questions about different things.

17.8 Exercises

1. **[W]** Explain why grid search over 5 hyperparameters with 5 values each needs $5^5 = 3{,}125$ training runs. Explain why that number would not shrink even if only 2 of the 5 hyperparameters mattered.
2. **[W]** State which search strategy is the reasonable default for: (a) tuning 2 hyperparameters on a model that trains in under a second; (b) tuning 8 hyperparameters on a neural network that takes 20 minutes per training run.
3. **[A]** Using Listing 17.5's measured results, compute the percentage-point gap between grid search's and random search's test AUC (0.8933 vs. 0.8942), and explain what about the grid's construction accounts for it.
4. **[A]** Using Procedure 17.4, sketch how you would add pruning to the Optuna study from this chapter's Worked Example if XGBoost's training reported intermediate validation AUC after every 10 boosting rounds, using `optuna.pruners.MedianPruner`.
5. **[E]** A colleague argues Optuna should always be used "because it's the most sample-efficient." Using this chapter's measured trial counts and times (Optuna matched random search's final score in 5 trials and 5.7 seconds, but its full 64-trial run took 171.5 seconds against random search's 18.5), write the two or three sentences you'd say back about when that advice is right and when it isn't.
6. **[E]** A team reports "we tuned hyperparameters and our validation AUC went from 0.91 to 0.97" as the headline result. Using this chapter's Validation Set Is Sacred section, write the question you'd ask before accepting that 0.97 figure.
7. **[R]** Read the Bergstra and Bengio (2012) abstract. Identify the specific dimensionality condition under which they show random search's advantage over grid search, and relate it to why this chapter's own six-hyperparameter, corners-only grid comparison showed the predicted gap.

17.9 Mental Model

Hyperparameter optimisation is like wine tasting: searching a space of configurations for the ones that perform well on a validation set, without being able to taste every bottle. Grid search tries every wine on the menu, in a fixed order. That's exactly its problem: a long menu takes forever to get through. Random search grabs bottles at random. Bayesian optimisation asks the sommelier to suggest the next bottle, based on which ones were liked before. That's an advantage when there are a lot of bottles and tasting each one is expensive. It doesn't matter at all when the menu is short.

18 Regularisation

A model reaches 98% accuracy on training data and 72% on unseen test data. That's a textbook overfit. The model learned the training set's noise instead of its underlying signal. Regularisation is the primary tool for fighting this. It works by penalising model complexity during training. That pushes the model towards simpler explanations that generalise better, rather than more complex ones that memorise.

18.1 Motivation

A model with enough degrees of freedom can perfectly memorise any training set. More data helps, but it often isn't available. You need to constrain how much the model is *allowed* to memorise. Regularisation adds a cost for complexity to the training objective. Instead of minimising prediction error alone, the model minimises prediction error plus a penalty for large or complex weights. Penalty terms for linear models, dropout for neural networks, depth limits for decision trees, early stopping for iterative training: all forms of regularisation. The form changes by model type. The underlying principle is the same across all of them.

18.2 Exposition

18.2.1 The Bias-Variance Trade-off

Chapter 11's Bias-Variance Trade-Off section introduced this decomposition in full: Total Error = Bias^2 + Variance + Noise. Regularisation trades a small increase in bias for a larger reduction in variance. The result is lower total error on unseen data.

18.2.2 Diagnosing with Learning Curves

The bias-variance trade-off is conceptual. A learning curve is how to see which side of it a model is on, before choosing whether to regularise, simplify, or collect more data. Train the model on increasing subsets of the training set (10%, 20%, and so on). At each size, record both training error and validation error, then plot both against training-set size. This is a different diagnostic from Chapter 4's loss curve, which plots error against training *epoch*

to check whether a model is converging. A learning curve asks a different question: would more data help?

If both curves converge to a similarly high error as training-set size grows, that's high bias. The model is too simple for the signal in the data. No amount of extra data will help. A more flexible model or better features is the fix. If training error stays low while validation error stays higher, with a gap that persists (though usually narrows) as training-set size grows, that's high variance. The model has the capacity to overfit. More data, or more regularisation, is the fix.

Listing 18.1 Learning curves for an underfit model (left) and an overfit model (right) on the same data

```
import numpy as np
from sklearn.model_selection import learning_curve
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

rng = np.random.RandomState(42)
X = rng.uniform(0, 10, 400).reshape(-1, 1)
y = 3 * np.sin(1.5 * X.ravel()) + rng.normal(0, 0.5, 400)

train_sizes = np.linspace(0.1, 1.0, 10)
fig, axes = plt.subplots(1, 2, figsize=(12, 4.5))

for ax, model, title in [
    (axes[0], LinearRegression(), 'High bias: LinearRegression on a sine curve'),
    (axes[1], DecisionTreeRegressor(),
     'High variance: unconstrained DecisionTreeRegressor'),
]:
    ts, train_scores, val_scores = learning_curve(
        model, X, y, train_sizes=train_sizes,
        cv=5, scoring='neg_root_mean_squared_error',
    )
    ax.plot(ts, -train_scores.mean(axis=1), color='#4E79A7', marker='o',
            label='Training error')
    ax.plot(ts, -val_scores.mean(axis=1), color='#E15759', marker='s',
            linestyle='--', label='Validation error')
    ax.set_xlabel('Training set size')
    ax.set_ylabel('RMSE')
    ax.set_title(title)
    ax.legend()

plt.tight_layout()
plt.show()
```



Figure 18.1: Learning curves on the same noisy sine-curve data: a straight line (left, high bias) and an unconstrained decision tree (right, high variance).

Figure 18.1 shows both failure modes side by side. On the left, `LinearRegression`'s training RMSE starts a touch lower, around 2.01 with the smallest training slice, then rises and settles alongside validation RMSE in a tight band, roughly 2.19 to 2.25, for the rest of the curve. Neither curve improves as more data arrives, and there's no consistent gap between them: the signature of high bias, a model too simple to represent a sine curve no matter how much data it sees. On the right, the unconstrained decision tree's training RMSE sits at nearly zero throughout. It memorises every training point exactly. Validation RMSE starts at 0.938, drops to around 0.71-0.75, and stays there: a gap that narrows a little, as the text above predicts, but never closes. That's the signature of high variance.

18.2.3 L2 Regularisation (Ridge)

Ridge adds the sum of squared weights to the loss: $\text{Loss}_{\text{ridge}} = \text{Loss}(y, \hat{y}) + \alpha \sum w_i^2$, where α controls penalty strength. This pushes gradient descent towards smaller weights across the board. Ridge shrinks every weight towards zero, but keeps every feature in the model. It's the right choice when most features are believed useful, but noisy.

18.2.4 L1 Regularisation (Lasso)

Lasso adds the sum of absolute weights instead: $\text{Loss}_{\text{lasso}} = \text{Loss}(y, \hat{y}) + \alpha \sum |w_i|$. The geometric reason Lasso behaves differently from Ridge is subtle. The L1 penalty's constraint region has corners exactly on the axes. When the constrained solution lands on one of those corners, a weight is driven to exactly zero. The L1 penalty has no gradient at zero, so scikit-learn fits Lasso by coordinate descent, not gradient descent. Lasso therefore produces

sparse solutions. It zeros out irrelevant features, making it a form of automatic feature selection that Ridge structurally cannot do.

18.2.5 Elastic Net

Elastic Net combines both penalties: $\text{Loss}_{\text{enet}} = \text{Loss}(y, \hat{y}) + \alpha(\rho \sum |w_i| + \frac{1}{2}(1-\rho) \sum w_i^2)$, as scikit-learn writes it. scikit-learn also scales the data-fit term by $1/(2n)$ for Lasso and Elastic Net, but not for Ridge. So the same α means a different strength in each of the three. It gets Lasso's sparsity while handling correlated features more gracefully. Lasso tends to pick one feature from a correlated group arbitrarily and zero out the rest. Elastic Net tends to keep or drop correlated groups together instead.

18.2.6 Dropout (Neural Networks)

During each training step, dropout randomly zeroes a fraction of neurons, preventing them from co-adapting into a fixed, fragile pattern. Any given neuron has to be independently useful, not just useful as part of one specific combination. During training, frameworks such as PyTorch scale the surviving activations up by $1/(1-p)$, where p is the dropout rate. That keeps the expected activation the same, so at inference every neuron is active and no scaling is needed. A dropout rate of 0.2-0.5 is typical. It's one of the most commonly used regularisers for deep networks.

18.2.7 Early Stopping

Monitor validation loss during training. Stop once it hasn't improved for a set number of epochs, the patience, even while training loss keeps falling. Validation loss is noisy, so a single rise isn't enough. Patience catches the point where overfitting has set in. It's simple, effective, and nearly free in compute terms. It's just stopping a run early.

18.3 Trade-offs and Limits

Too much regularisation (α too high) underfits. The model can't capture the real signal, and both training and test performance suffer together. Too little (α too low, or none) overfits. Training error is low but test error is high. A widening gap between the two is the real symptom. The sweet spot is found by cross-validated search over α: `RidgeCV`, `LassoCV`, or the general hyperparameter search machinery from Chapter 17.

18.4 Procedures

Procedure 18.1 (Choose Between Ridge, Lasso, and Elastic Net). *When to use:* selecting a regularisation strategy for a linear model.

Listing 18.2 Decision path for Ridge vs. Lasso vs. Elastic Net

```
# Want automatic feature selection (some weights -> exactly 0)?
# No  -> Ridge (L2): keeps all features, shrinks them
# Yes -> features correlated with each other?
#        Yes -> Elastic Net (handles correlated groups better)
#        No  -> Lasso (L1)
```

Quick rule: start with Ridge; try Lasso if many features are suspected irrelevant; use Elastic Net when features are highly correlated.

Procedure 18.2 (Apply Ridge with Cross-Validated Alpha). *When to use:* the default regularised-regression starting point.

Listing 18.3 RidgeCV with a log-spaced alpha grid

```
from sklearn.linear_model import RidgeCV
import numpy as np

ridge_cv = RidgeCV(alphas=np.logspace(-3, 3, 100), cv=5)
ridge_cv.fit(X_train_scaled, y_train)
print(f'Best alpha: {ridge_cv.alpha_}')
```

Procedure 18.3 (Apply Lasso and Count Eliminated Features). *When to use:* suspected irrelevant features, or automatic feature selection is wanted alongside regularisation.

Listing 18.4 LassoCV and inspecting which coefficients were zeroed

```
from sklearn.linear_model import LassoCV

lasso_cv = LassoCV(
    alphas=np.logspace(-4, 1, 100), cv=5, max_iter=10000
)
lasso_cv.fit(X_train_scaled, y_train)
n_zero = (lasso_cv.coef_ == 0).sum()
```

```
n_total = len(lasso_cv.coef_)
print(
    f'{n_zero} of {n_total} features eliminated, '
    f'best alpha={lasso_cv.alpha_}'
)
```

Common mistake: forgetting to scale features before regularising. An unscaled feature on a much larger numeric range gets penalised unfairly relative to a small-scale one, since the penalty is applied to the raw weight magnitude. Always `StandardScaler()` first.

Procedure 18.4 (Apply Regularised Logistic Regression). *When to use:* classification with the same overfitting risk.

Listing 18.5 LogisticRegressionCV for classification

```
from sklearn.linear_model import LogisticRegressionCV

# C is the inverse of alpha: smaller C = more regularisation
lr_l2 = LogisticRegressionCV(
    Cs=20, cv=5, max_iter=1000,  # L2 is the default penalty
)
lr_l2.fit(X_train_scaled, y_train)
```

Procedure 18.5 (Diagnose Bias vs. Variance with a Learning Curve). *When to use:* deciding whether "get more data" or "add regularisation, or a simpler model" is the right next step, before spending time or money on either.

Listing 18.6 Plotting training and validation error against training-set size

```
from sklearn.model_selection import learning_curve
import numpy as np

train_sizes, train_scores, val_scores = learning_curve(
    estimator, X_train, y_train,
    train_sizes=np.linspace(0.1, 1.0, 10),
    cv=5, scoring='neg_root_mean_squared_error',
)

train_rmse = -train_scores.mean(axis=1)
val_rmse = -val_scores.mean(axis=1)

plt.plot(train_sizes, train_rmse, label='Training error')
```

```
plt.plot(train_sizes, val_rmse, label='Validation error')
plt.xlabel('Training set size')
plt.ylabel('RMSE')
plt.legend()
```

Common mistake: reading the curve at a single training-set size instead of its overall shape. The diagnostic signal is whether the gap between the two curves narrows as size increases, not the error at any one point on its own.

18.5 Worked Example

Fitting a degree-15 polynomial to 60 noisy points generated from a simple quadratic ($y = 0.5x^2 - x + 2$ plus Gaussian noise), with a 70/30 train/test split, produces the textbook overfitting shape:

Listing 18.7 An unregularised degree-15 polynomial memorising noise

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression

rng = np.random.RandomState(42)
X = rng.uniform(-5, 5, 60).reshape(-1, 1)
y = 0.5 * X.ravel()**2 - X.ravel() + 2 + rng.normal(0, 2, 60)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

poly = PolynomialFeatures(degree=15, include_bias=False)
scaler = StandardScaler()
X_train_poly = scaler.fit_transform(poly.fit_transform(X_train))
X_test_poly = scaler.transform(poly.transform(X_test))

lr = LinearRegression().fit(X_train_poly, y_train)
```

Measured: unregularised, this model reaches training RMSE 1.412 but test RMSE 1.912, about 35% worse on data it hasn't seen. That's a real but modest overfitting gap, given how small this dataset is (42 training points). Applying `RidgeCV` (best $\alpha = 2.477$) brings test RMSE down to 1.461, *below* even its own training RMSE of 1.819. The regularised model

generalises better than it fits, the opposite of the unregularised model's pattern. `LassoCV` (best $\alpha = 0.076$) does comparably well on test RMSE (1.455, the best of the three), while additionally zeroing out 11 of the 15 polynomial-feature coefficients entirely. Lasso didn't just shrink the overfit away. It explicitly decided most of the degree-15 expansion's terms weren't worth keeping, leaving a much smaller model behind that generalises better than the 15-term original it started from, even without recovering the exact quadratic shape.

This is the bias-variance trade-off made concrete. Both Ridge and Lasso accept worse *training* fit than the unregularised model (RMSE 1.819 and 1.793 vs. 1.412), in exchange for better *test* performance (1.461 and 1.455 vs. 1.912). That's a deliberate increase in bias that gets a larger reduction in variance, exactly as the Bias-Variance Trade-off section predicts.

AWS SageMaker Equivalent

```
from sagemaker.linear_learner import LinearLearner

ll = LinearLearner(
    predictor_type='binary_classifier',
    l1=0.01,    # L1 regularisation strength
    wd=0.001,   # weight decay = L2 regularisation
)
```

When to prefer SageMaker Linear Learner: when the regularised linear model needs to be trained at a scale beyond a single machine's memory, or deployed as a managed endpoint without hand-building the serving infrastructure. For the polynomial-overfitting scale in this chapter's Worked Example, a local `scikit-learn` pipeline is simpler and faster.

18.6 History and Further Reading

Murphy's Introduction, Ch. 13, gives the full mathematical treatment of L1, L2, and Elastic Net (Murphy 2022). Burkov, Ch. 4, explains regularisation accessibly with visuals (Burkov 2019). Géron's *Hands-On Machine Learning*, Ch. 4, works through Ridge, Lasso, and Elastic Net side by side in code, including the geometric picture of why Lasso zeroes out coefficients and Ridge only shrinks them (Géron 2019). Bishop's *Pattern Recognition and Machine Learning*, Ch. 3, gives the probabilistic interpretation (L2 as a Gaussian prior, L1 as a Laplace prior) (Bishop 2006).

18.7 How This Connects

This chapter builds on Chapter 4's calculus and optimisation (regularisation modifies the loss function being optimised) and Chapter 13's cross-validation (needed to choose α properly).

It enables the dropout and weight-decay techniques Book 2, Chapter 28 applies to neural networks, where L2 regularisation reappears under the name "weight decay." It's often confused with feature selection (Chapter 15). L1 regularisation *does* perform feature selection as a side effect, but Chapter 15 covers the explicit, dedicated methods for it.

18.8 Exercises

1. [**W**] Explain in your own words why Ridge shrinks weights towards zero without ever setting them exactly to zero, while Lasso can.
2. [**W**] A model shows training RMSE 1.2 and test RMSE 3.1. State whether this is a bias problem or a variance problem, and name one regularisation technique that would help.
3. [**A**] Using Listing 18.7's measured numbers, compute the ratio of test RMSE to train RMSE for the unregularised model (1.912/1.412) and for the Ridge model (1.461/1.819), and explain what a ratio below 1.0 shows about the Ridge model's fit.
4. [**A**] Using Listing 18.4's pattern, explain why Lasso zeroing 11 of 15 polynomial coefficients is broadly consistent with the true underlying relationship being quadratic (degree 2, i.e. needing only 2 non-trivial polynomial terms beyond the intercept), even though a few extra nonzero terms remain, likely absorbing noise from a training set of only 42 points.
5. [**E**] A colleague sets `alpha=100` for a Ridge model "to be safe against overfitting" without cross-validating it. Using this chapter's Trade-offs section, explain what's likely to go wrong.
6. [**E**] A teammate regularises a model with features on wildly different scales (one column in the thousands, another between 0 and 1) without scaling first. Using Procedure 18.3's common mistake, predict what will happen to the two features' effective regularisation strength.
7. [**R**] Read the Bishop (Pattern Recognition and Machine Learning) Ch. 3 treatment of L2 as a Gaussian prior. Explain, in your own words, how choosing a smaller prior variance corresponds to a larger α in the Ridge loss function.

18.9 Mental Model

Regularisation is Occam's Razor applied to machine learning. *Among models that explain the data equally well, prefer the simpler one.* The penalty term is a formal, tunable statement of that preference. α controls exactly how strongly simplicity is preferred over fit. Turn the

dial too low and the model memorises noise. Turn it too high and the model can't see the signal.

19 Unsupervised Learning

Every model up to this chapter has required labels. Someone had to mark each example "malignant" or "benign," "fraud" or "not fraud." Labels are expensive. A radiologist reviewing scans, an analyst labelling transactions: both cost real time and money. For many real questions, like which customers resemble each other or whether a log entry looks unusual, there are no labels, only raw data. Unsupervised learning finds structure without labels. It's less glamorous than supervised learning. But it's often where the most interesting discoveries happen.

19.1 Motivation

There are two main tasks here. *Clustering* groups similar things together without being told what the groups are. Use cases include customer segmentation, document grouping, and finding natural categories in any dataset. *Anomaly detection* finds things that don't fit. Use cases include fraud detection on new accounts with no labelled fraud yet, network intrusion detection, and manufacturing defect detection. Both tasks share an underlying question: what does it mean to be "similar," and therefore "unusual," in this space? The answer depends on how distance or density is defined.

19.2 Exposition

19.2.1 K-Means Clustering

K-Means is the clustering workhorse. Randomly initialise K centroids. A centroid is just the cluster's mean point: the average of every feature, taken across every point currently assigned to it. Assign each point to its nearest centroid. Recompute each centroid as the mean of its assigned points. Repeat until assignments stop changing. It converges to a local minimum, which is why it is often run several times from different starts, keeping the best result (`n_init=10` in scikit-learn; the default `'auto'` runs once with k-means++ initialisation). It's fast, scalable, and easy to interpret. But it needs K decided in advance, and assumes clusters are roughly spherical and similar in size.

Specifying K is sometimes easy. A cancer-diagnosis dataset has an obvious $K = 2$: malignant or benign, fixed by the biology being studied, not a choice anyone has to make. Other times

it's close to impossible to know in advance. Segmenting an e-commerce platform's customers could reasonably use 3 broad tiers (high, medium, low value) or 300 narrow micro-segments, one per campaign. Nothing about the data itself says which is right. The correct K depends on what the business plans to do with the answer, not on the customers themselves.

That spherical, similar-size assumption fails on the wrong kind of data, as this chapter's Worked Example shows directly. Choosing K usually combines two tools. The elbow method plots inertia (the summed squared distance from each point to its assigned centroid) against K and looks for diminishing returns. The second tool is the silhouette score (-1 to 1, higher better, measuring how well each point fits its own cluster against the next-best alternative).

19.2.2 Gaussian Mixture Models

K-Means assigns every point to exactly one cluster and assumes clusters are roughly spherical. A **Gaussian Mixture Model (GMM)** relaxes both assumptions. It models the data as if it were generated by a mixture of several Gaussian distributions, each with its own mean and covariance, and fits them with Expectation-Maximisation. The E-step computes each point's probability of belonging to each Gaussian, a soft assignment, not a hard 0/1 label. The M-step updates each Gaussian's mean, covariance, and weight to better fit those soft assignments, repeating until convergence. That's structurally the same assign-then-recompute loop K-Means uses, made probabilistic.

Because each component has its own covariance matrix, GMM can fit elongated or rotated elliptical clusters, not just K-Means's implicitly circular ones. And because membership is a probability rather than a label, a point near a cluster boundary can have, say, a 60/40 split between two components. That's often a truer description of ambiguous real data than forcing a choice. GMM is a proper probabilistic model with a likelihood, so choosing the number of components can use BIC or AIC instead of K-Means's elbow-and-silhouette combination above. Both are a single score that rewards a good fit but penalises extra components. The number of components that gets the lowest score is the one to pick. BIC (Bayesian Information Criterion) increases that penalty as the dataset grows larger. AIC (Akaike Information Criterion) keeps the penalty fixed no matter the size. In practice, try both. Plot the score against the number of components, and look for the point where the score stops dropping: the same diminishing-returns idea as the elbow method, just applied to a different number. The added flexibility isn't free. Fitting a full covariance matrix per component costs more compute than K-Means's single centroid per cluster. And GMM can still converge to a poor local optimum, the same failure mode K-Means has.

19.2.3 DBSCAN

DBSCAN needs no K decided in advance. It defines clusters as dense regions separated by sparse ones, using two parameters: `eps` (the neighbourhood radius) and `min_samples`

(how many neighbours within `eps` make a point a "core point"). Core points and everything density-reachable from them form a cluster. Anything left over is labelled noise (`-1`), which doubles as automatic outlier detection. DBSCAN finds arbitrarily shaped clusters that K-Means structurally cannot. That comes at the cost of sensitivity to its two parameters and difficulty with clusters of varying density.

19.2.4 Hierarchical Clustering

Hierarchical (agglomerative) clustering builds a dendrogram: a tree showing the order clusters were merged, which can be cut at any height to get any number of clusters. It's useful for exploratory work when the right number of clusters isn't known in advance. But its time and memory costs grow at least quadratically with the number of rows ($O(n^3)$ time for the naive algorithm), which makes it impractical past roughly 10,000 rows.

19.2.5 Isolation Forest

Isolation Forest works on the idea that anomalies are few and different, so they're easy to isolate. Build a forest where each tree splits on a randomly chosen feature and split value. A genuine anomaly, sitting apart from the bulk of the data, needs far fewer random splits to end up alone in its own partition than a typical point does. The anomaly score is based on average path length to isolation across all trees. A short path means easy isolation means likely anomaly. It scales well and works in high dimensions. But results vary run to run (it's inherently random), and the `contamination` parameter (expected anomaly fraction) has to be set from real domain knowledge.

19.3 Trade-offs and Limits

Without ground-truth labels, there's no objective way to know if the "right" clusters were found. Silhouette and similar internal metrics measure geometric compactness and separation, not business usefulness. A clustering with a mediocre silhouette score can still be the most actionable one, if a domain expert confirms the groups map to real, distinct behaviour. Anomaly detection has the same problem. "Anomaly" is domain-specific: a large purchase is normal for one customer and suspicious for another. Results are very sensitive to the contamination assumption. Ground-truth labels, when available, always beat unsupervised detection. Unsupervised methods are where you turn when labels don't exist yet.

19.4 Procedures

Procedure 19.1 (Choose the Number of Clusters). *When to use:* before running K-Means or any fixed-K algorithm.

Listing 19.1 Elbow method and silhouette score together

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

inertias, silhouettes = [], []
for k in range(2, 11):
    km = KMeans(
        n_clusters=k, random_state=42, n_init=10
    ).fit(X_scaled)
    inertias.append(km.inertia_)
    silhouettes.append(silhouette_score(X_scaled, km.labels_))
```

Common mistake: picking K purely from an ambiguous elbow chart. Cross-check with the silhouette score, and, more important than either, check whether the resulting clusters make sense to someone who knows the domain.

Procedure 19.2 (Run K-Means, GMM, DBSCAN, and Agglomerative Clustering). *When to use:* a feature matrix with rows to group.

Listing 19.2 Four clustering approaches for four different assumptions

```
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.mixture import GaussianMixture

km = KMeans(
    n_clusters=4, random_state=42, n_init=10
).fit_predict(X_scaled)
gm = GaussianMixture(
    n_components=4, random_state=42
).fit_predict(X_scaled)
db = DBSCAN(eps=0.5, min_samples=5).fit_predict(X_scaled)
agg = AgglomerativeClustering(
    n_clusters=4, linkage='ward'
).fit_predict(X_scaled)
```

Common mistake: calling `DBSCAN(eps=0.5, ...)` on unscaled features. `eps` is a raw distance threshold, so a feature ranging 0-100,000 will dominate the distance calculation over one ranging 0-1 entirely.

Procedure 19.3 (Detect Anomalies with IsolationForest). *When to use:* flagging unusual rows without enough labelled anomalies to train a supervised classifier.

Listing 19.3 IsolationForest with a domain-informed contamination rate

```
from sklearn.ensemble import IsolationForest

iso = IsolationForest(
    n_estimators=100, contamination=0.05, random_state=42
)
labels = iso.fit_predict(X_scaled)   # 1 = normal, -1 = anomaly
scores = iso.score_samples(X_scaled) # lower = more anomalous
```

Common mistake: treating `contamination` as a free parameter to tune for a "better-looking" result. It should reflect an actual prior belief about how much of the data is anomalous: tuning it to hit a desired output count is circular and either hides real anomalies or manufactures fake ones.

Procedure 19.4 (Validate Clustering Quality Without Ground Truth). *When to use:* after clustering, before trusting or shipping the result.

Listing 19.4 Internal clustering-quality metrics

```
from sklearn.metrics import (
    silhouette_score,
    davies_bouldin_score,
    calinski_harabasz_score,
)

# -1 to 1, higher better
sil = silhouette_score(X_scaled, labels)
# 0 to inf, lower better
db_score = davies_bouldin_score(X_scaled, labels)
```

19.5 Worked Example

Fitting K-Means to two interleaving crescent-shaped clusters (`make_moons`, 300 points, moderate noise) demonstrates the spherical-cluster assumption failing directly, not just in theory:

Listing 19.5 K-Means vs. GMM vs. DBSCAN on non-spherical clusters

```
km = KMeans(n_clusters=2, random_state=42, n_init=10).fit(X_scaled)
gm = GaussianMixture(n_components=2, random_state=42).fit(X_scaled)
db = DBSCAN(eps=0.3, min_samples=5).fit(X_scaled)
```

Measured, scored against the true crescent labels using the Adjusted Rand Index (1.0 = perfect agreement, 0 = random): K-Means achieves ARI 0.479. It finds *a* split of the data, since it always finds some split. But that split cuts across the two crescents rather than separating them, because K-Means divides space with straight boundaries halfway between centroids, so every cluster it finds is a convex region, and the true clusters are not remotely convex. A Gaussian Mixture Model, fit with two components on the same data, lands on the exact same ARI, 0.479, despite its elliptical covariance being strictly more flexible than K-Means's fixed circular one. Here, that extra flexibility doesn't help at all. The underlying problem isn't circularity, it's convexity. Two crescents can't be well-approximated by two ellipses any more than by two circles. GMM shares K-Means's assumption that each cluster is a single convex blob, so both algorithms end up drawing essentially the same dividing line. DBSCAN, with `eps=0.3` chosen from a k-distance plot, achieves ARI 0.987, with only 2 of 300 points labelled noise. That's close to a perfect recovery of both crescents. This isn't DBSCAN being "better" in general. It's DBSCAN's density-based definition of a cluster matching this data's actual shape, while K-Means's and GMM's centroid-based definitions structurally cannot.

A separate anomaly-detection problem is deliberately harder. It has 500 points drawn from a standard 2D normal distribution, plus 25 outliers scattered uniformly across a much wider range. An IsolationForest with `contamination=0.05` (matching the true $25/525 \approx 4.8\%$ anomaly rate) flags 27 points as anomalies, with precision 0.852 and recall 0.920. It catches most of the real anomalies (23 of 25), while incorrectly flagging a small handful of ordinary points that happened to land unusually far from the bulk of the normal distribution just by chance. That's a realistic result, not a perfect one. It's a useful reminder that even a well-calibrated `contamination` estimate doesn't produce perfect precision or recall on real, noisy data.

AWS SageMaker Equivalent

```
from sagemaker import KMeans as SageMakerKMeans
from sagemaker import RandomCutForest

kmeans_estimator = SageMakerKMeans(
    role=role,
    instance_count=1,
    instance_type='ml.c5.xlarge',
    k=4,
    output_path=f's3://{bucket}/kmeans-output',
)
kmeans_estimator.fit(
    kmeans_estimator.record_set(X_train.astype('float32'))
)

rcf_estimator = RandomCutForest(
    role=role,
    instance_count=1,
    instance_type='ml.c5.xlarge',
    num_trees=100,
    num_samples_per_tree=256,
)
rcf_estimator.fit(
    rcf_estimator.record_set(X_train.astype('float32'))
)
```

When to prefer SageMaker's built-in K-Means/Random Cut Forest: once clustering or anomaly detection needs to run on data too large to fit in memory on one machine, or as a scheduled production pipeline with monitoring and retraining. For exploratory work on data that fits in memory, like this chapter's Worked Example, sklearn is faster to iterate with.

19.6 History and Further Reading

Murphy's Introduction, Ch. 21-22, gives the full treatment of K-Means, Gaussian Mixture Models, and DBSCAN (Murphy 2022). Burkov, Ch. 9, gives an accessible unsupervised learning overview (Burkov 2019). Géron's *Hands-On Machine Learning*, Ch. 9, covers K-Means, DBSCAN, and Gaussian Mixture Models with the Scikit-Learn code for each, a practical companion to this chapter's algorithms (Géron 2019). Liu et al.'s original Isolation Forest paper gives the full algorithm and its complexity analysis (Liu et al. 2008).

19.7 How This Connects

This chapter builds on Chapter 6's Python ecosystem and Chapter 9's EDA (understanding data before clustering it), and on Chapter 15's dimensionality reduction, which PCA/UMAP use to visualise high-dimensional clusters in 2D. It enables Book 2, Chapter 31's retrieval-augmented generation, where document clustering and embedding-based similarity underpin how vector databases organise retrieval. It's often confused with semi-supervised learning (some labels exist) and self-supervised learning (a model creates its own supervision signal, central to foundation model pretraining from Book 2, Chapter 29 onward). Both are related but distinct ideas, apart from the fully-unlabelled setting this chapter covers.

19.8 Exercises

1. **[W]** Explain in one or two sentences why K-Means needs K decided in advance while DBSCAN does not.
2. **[W]** State which algorithm (K-Means or DBSCAN) is the better default for each: (a) a large, roughly spherical, well-separated set of customer segments; (b) an unknown number of irregularly shaped clusters with some genuine noise points mixed in.
3. **[A]** Using Listing 19.5's measured ARI scores (K-Means 0.479, DBSCAN 0.987), explain in your own words why a higher silhouette score for K-Means's own clustering wouldn't necessarily have revealed this failure: silhouette measures compactness *given* the clustering K-Means chose, not whether that clustering matches the true structure.
4. **[A]** Using Listing 19.3's measured results (27 flagged, precision 0.852, recall 0.920), compute the number of false positives and false negatives, and state which of the two errors this particular run made more of.
5. **[E]** A customer wants to segment their customer base and reports "we found 6 clusters with a great silhouette score." Using this chapter's Trade-offs section, write the follow-up question you'd ask before treating those 6 clusters as actionable business segments.
6. **[E]** A fraud team sets IsolationForest's `contamination=0.20` "because that gives us the cleanest-looking separation in our validation plots." Using Procedure 19.3's common mistake, explain why this reasoning is circular.
7. **[R]** Read the original Isolation Forest paper's (Liu et al., 2008) abstract. Identify the specific computational-complexity claim it makes relative to distance-based anomaly detection methods, and explain how that relates to this chapter's "scales well, works in high dimensions" summary.

19.9 Mental Model

Clustering is map-making: drawing a map of the data space, grouping nearby points into regions. The map isn't the territory. There's no single "correct" clustering. Only clusterings useful for a given purpose. Different algorithms draw different maps, by definition. This chapter's moons example is exactly what it looks like when the map's assumptions don't match the territory's actual shape.

20 Time Series Analysis

Every model we've met so far has assumed rows are independent and identically distributed. Shuffle them and nothing changes. A random 80/20 split is fine. Ordinary k-fold cross-validation works as-is. Time series data breaks that assumption. Today's value depends on yesterday's. December's sales depend on last December's. A random split leaks the future into training and quietly inflates every reported metric. The model gets to peek at values from after the point it's supposed to be predicting. This chapter is the toolkit built around the thing we've so far ignored: order.

20.1 Motivation

A time series is a sequence of values indexed by time. Almost every real series is a mix of three components: trend (the long-run direction), seasonality (a pattern repeating at a fixed, known period), and residual noise (whatever's left once trend and seasonality are removed). Classical decomposition splits a series into exactly these pieces, additively ($y = \text{trend} + \text{seasonal} + \text{residual}$), or multiplicatively when the seasonal swing grows with the level of the series. Once this is understood, the modelling task becomes clear. Forecast the trend. Forecast the seasonal pattern. Treat the residual as irreducible uncertainty.

20.2 Exposition

20.2.1 Stationarity

Most classical forecasting models (ARIMA and its relatives) assume the series is stationary. Its mean, variance, and autocorrelation don't change over time, so the bit of history a model is fit to generalises to the next stretch. Most real series aren't stationary on their own. They trend, and their seasonal amplitude often grows too. Differencing (subtracting each value from the one before it, $y'_t = y_t - y_{t-1}$) removes a linear trend. Seasonal differencing ($y'_t = y_t - y_{t-m}$) removes seasonality at period m. The Augmented Dickey-Fuller (ADF) test checks whether differencing worked. Its null hypothesis is that the series has a unit root (is non-stationary). A small p-value (usually below 0.05) lets that null be rejected.

20.2.2 Autocorrelation and Partial Autocorrelation

Autocorrelation (ACF) at lag k measures how similar y_t is to y_{t-k}. Partial autocorrelation (PACF) measures the same relationship with the effect of all shorter lags already removed: the direct connection, not the connection mediated through intervening lags. These aren't just diagnostics. This is how model order gets chosen. A sharp PACF cutoff after lag p suggests an AR(p) term. A sharp ACF cutoff after lag q suggests an MA(q) term. A slowly decaying ACF is the signature of non-stationarity: a signal to difference before fitting anything.

20.2.3 ARIMA: AR, I, MA

ARIMA(p, d, q) combines three ideas. AR(p), autoregressive: today's value is a linear combination of the p previous values plus noise, capturing momentum. I(d), integrated: difference the series d times to make it stationary before fitting. $d = 1$ handles a linear trend. $d = 2$ handles a trend that's accelerating (rare in practice). MA(q), moving average: today's value is a linear combination of the q previous forecast *errors*, capturing short-lived shocks that take a few periods to wash out. SARIMA adds a second, seasonal (P, D, Q, m) block on top of the same three ideas, applied at the seasonal lag m instead of lag 1. Use it whenever ACF/PACF or the decomposition shows a repeating pattern the plain ARIMA terms can't capture.

20.2.4 Exponential Smoothing

A simpler approach, from a simpler time. Or the Honey Badger of forecasting models, taking on beasts much bigger than itself. You decide! The idea is simple: forecast the next value as a weighted average of past values, with weights decaying exponentially into the past. Simple exponential smoothing handles level only. One smoothed average, nudged a little towards each new value as it arrives. Holt's method (Holt 1957) adds a second smoothed term for trend, so the forecast can keep rising or falling instead of flattening out at the last level. Holt-Winters (Winters 1960) adds a third smoothed term on top of that, one for seasonality repeating at a fixed period. Three components, three running averages: one for level, one for trend, one for season, each updated a little every time a new value comes in. It's cheaper to fit than ARIMA and a strong baseline. Always fit one before reaching for anything more complex.

20.2.5 Beyond Classical Statistics: ML and Deep Learning

Classical models work directly on a single series and its own history. A different, increasingly common approach treats forecasting as supervised regression. Engineer lag features (y_{t-1},

y_{t-7}, y_{t-30}, …) and rolling statistics as columns, then fit a gradient-boosted tree (Chapter 12) to predict y_t from them. This folds in external features (price, weather, promotions) that ARIMA has no natural way to use, at the cost of losing ARIMA's built-in uncertainty intervals. Prophet sits in between: an additive trend-plus-seasonality model fit via curve-fitting rather than the ARIMA machinery, popular for its robustness to missing data and minimal tuning. Deep learning models (LSTMs, temporal convolutional networks, transformer-based forecasters) can model many related series jointly and capture long-range or nonlinear dependencies classical models can't, at the cost of needing far more data and tuning. They're usually overkill for a single series with a clear seasonal pattern.

20.3 Trade-offs and Limits

ARIMA is more flexible than exponential smoothing, but needs a stationary series and careful order selection. Exponential smoothing is simpler, needs less tuning, and is often just as accurate in practice. Fit both. Keep whichever generalises better on held-out data. Classical models give prediction intervals for free (well calibrated only when the model's assumptions hold) and work with very little data. Lag-feature ML models scale better to many related series and easily incorporate external regressors, but need careful leakage-free cross-validation and don't naturally produce calibrated uncertainty. Use multiplicative decomposition when the seasonal swing grows with the series' level. Use additive when it stays roughly constant. This is usually obvious when plotted.

20.4 Procedures

Procedure 20.1 (Split Time Series Data Correctly). *When to use:* every time, never a random split or plain k-fold on time-ordered data.

Listing 20.1 Chronological holdout and TimeSeriesSplit

```
from sklearn.model_selection import TimeSeriesSplit

horizon = 28
train, test = df.iloc[:-horizon], df.iloc[-horizon:]

tscv = TimeSeriesSplit(n_splits=5, test_size=28)
for train_idx, test_idx in tscv.split(df):
    # Each fold's test block is always chronologically
    # after its train block
    pass
```

Common mistake: using `train_test_split(shuffle=True)` or plain `KFold` on time-indexed data. This lets the model train on future values to predict the past, producing metrics that look great in development and fail in production.

Procedure 20.2 (Test and Achieve Stationarity). *When to use:* before fitting any ARIMA/SARIMA model.

Listing 20.2 ADF test and differencing

```
from statsmodels.tsa.stattools import adfuller

def is_stationary(series, alpha=0.05):
    return adfuller(series.dropna())[1] < alpha

diff1 = series.diff()              # removes a linear trend
seasonal_diff = series.diff(7)     # removes weekly seasonality
```

Common mistake: over-differencing a series that's already stationary. This introduces artificial oscillation and inflates variance. Re-run the ADF test after each differencing step rather than differencing "just in case."

Procedure 20.3 (Choose ARIMA/SARIMA Orders). *When to use:* after confirming the differencing order needed for stationarity.

Listing 20.3 Visual and automated order selection

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(diff1.dropna(), lags=30)
plot_pacf(diff1.dropna(), lags=30)

import pmdarima as pm
auto_model = pm.auto_arima(
    series, seasonal=True, m=7, stepwise=True, trace=True
)
```

Procedure 20.4 (Evaluate a Forecast Against a Naive Baseline). *When to use:* after any forecast, before deciding a model is good enough to ship.

Listing 20.4 Always compare against the naive baseline

```
def forecast_metrics(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    rmse = mean_squared_error(y_true, y_pred) ** 0.5
    return {'MAE': mae, 'RMSE': rmse}

# Or shift(7) for weekly seasonal naive
naive = y_true.shift(1).bfill()
```

Common mistake: skipping the naive baseline. This chapter's own Worked Example shows exactly why that's a mistake: a fitted model can lose to "repeat the last value," and there's no way to know that without checking.

20.5 Worked Example

A synthetic two-year daily series (slow upward trend plus a smooth annual sine-wave seasonal component plus moderate Gaussian noise) makes stationarity and forecast evaluation concrete. The raw series fails the ADF test outright: $p = 0.682$, nowhere near significant, cannot reject non-stationarity. That's exactly as expected given its visible trend. First-differencing fixes this clearly: $p \approx 3 \times 10^{-27}$, far below 0.05.

Decomposing the series confirms what it was built from: trend rising from about 109 to 127 over the two years, seasonal amplitude of roughly 16 units. The pieces are recovered correctly by construction, a useful sanity check before trusting decomposition on a real series where the ground truth isn't known in advance.

Listing 20.5 ARIMA forecast vs. the naive baseline, over a 28-day horizon

```
model = SARIMAX(
    train,
    order=(2, 1, 2),
    enforce_stationarity=False,
    enforce_invertibility=False,
)
fitted = model.fit(disp=False)
pred = fitted.get_forecast(steps=28).predicted_mean

naive = pd.Series([train.iloc[-1]] * 28, index=test.index)
```

Measured, forecasting the final 28 days: an ARIMA(2,1,2) model reaches MAE 7.228 and RMSE 8.664. The naive baseline (repeating the last training value for all 28 days) reaches MAE 4.602 and RMSE 5.713, *beating* the fitted ARIMA model on both metrics. This is not a mistake in the setup. It's a real, common outcome worth understanding rather than hiding. Over a 28-day horizon against a 365-day seasonal period and a slow linear trend, the series barely moves in expectation. "Stay near the current level" is already close to the right answer. ARIMA's extra flexibility, fitting autoregressive and moving-average structure, is mostly available to fit short-term noise correlation it has no real ability to predict out-of-sample. The model isn't wrong to be more complex. The complexity doesn't help at this particular horizon, on this particular series. This is exactly why Procedure 20.4 insists on the naive baseline every time. Without it, a perfectly reasonable-looking ARIMA fit with plausible-looking diagnostics would have shipped without anyone noticing it was worse than doing nothing extra at all.

AWS SageMaker Equivalent

```
import boto3

forecast = boto3.client('forecast')
# Amazon Forecast automatically tries multiple algorithms
# (ARIMA, ETS, DeepAR+, Prophet, CNN-QR) and selects the
# best for the data via AutoML.
# predictor = forecast.create_auto_predictor(
#     PredictorName='demand-forecast',
#     ForecastHorizon=28,
#     ForecastFrequency='D',
#     DataConfig={'DatasetGroupArn': dataset_group_arn},
# )
```

When to prefer Amazon Forecast over hand-rolled ARIMA: with many related series (hundreds of SKUs, stores, or sensors), where Forecast trains global models across all series and handles algorithm selection automatically. That includes, implicitly, the naive-baseline comparison this chapter insists on doing by hand. For a single series or a handful, `statsmodels`/`pmdarima` is faster to iterate on and easier to explain to stakeholders.

Amazon Forecast closed to new customers in 2024. Existing accounts can still use it, and AWS points new users to SageMaker Canvas's time-series forecasting or SageMaker's built-in DeepAR algorithm. The same advice applies to those.

20.6 History and Further Reading

Hyndman and Athanasopoulos's *Forecasting: Principles and Practice* is the standard modern reference, covering ARIMA, exponential smoothing, and ML-based forecasting side by side (Hyndman and Athanasopoulos 2018). Murphy's Advanced Topics, Ch. 29, covers the state-space-model machinery underneath ARIMA and exponential smoothing (Murphy 2023). Box, Jenkins, Reinsel, and Ljung's *Time Series Analysis: Forecasting and Control* is the original, rigorous treatment of the Box-Jenkins approach to ARIMA (Box et al. 2015).

20.7 How This Connects

This chapter builds on Chapter 13's cross-validation, though time-series CV works differently: folds must respect chronological order via `TimeSeriesSplit`, never training on the future to predict the past. It builds on Chapter 9's EDA, since decomposition and ACF/PACF plots are exploratory analysis specialised for sequential data. It connects to Chapter 19's unsupervised learning, since anomaly detection on a time series usually starts by modelling the expected pattern with this chapter's tools and then flagging large residuals. It enables Book 2, Chapter 25's monitoring, where drift detection is, underneath, a time series problem: is this metric's distribution changing over time? It's often confused with sequential or panel data in general. Time series specifically means one variable (or a small related set) observed at regular intervals where *order* carries information.

This chapter closes Book 1. Book 2 opens with responsible AI and data governance, and from there moves into the neural-network and large-language-model territory that increasingly defines what a Solutions Architect is asked to build.

20.8 Exercises

1. [**W**] Explain in your own words why a random 80/20 train/test split is invalid for time series data but fine for the datasets used in earlier chapters.
2. [**W**] State what a slowly-decaying ACF (no sharp cutoff at any lag) tells you about a series, and what action it suggests.
3. [**A**] Using this chapter's measured ADF p-values (0.682 raw, roughly 3×10^{-27} differenced), explain what "reject the null hypothesis" means concretely for the differenced series, using the 0.05 significance threshold.
4. [**A**] Using Listing 20.5's measured results (ARIMA MAE 7.228, naive MAE 4.602), compute the percentage by which the naive baseline outperforms ARIMA, and propose one change to the forecast horizon (longer or shorter) that this chapter's explanation would predict might reverse that result.

5. [**E**] A team reports "our ARIMA model achieves MAE of 7.2 on 28-day demand forecasts" as evidence it's ready to ship. Using this chapter's Worked Example, write the question you'd ask before agreeing.
6. [**E**] A colleague computes a 7-day rolling mean feature for a lag-feature forecasting model using `.rolling(7).mean()` without first calling `.shift(1)`. Using Procedure 20.1's common mistake about leakage, explain what's wrong.
7. [**R**] Read the Hyndman and Athanasopoulos chapter on forecast evaluation (fpp3, "Evaluating forecast accuracy"). Identify one accuracy measure they discuss that this chapter's reference material does not (beyond MAE, RMSE, MAPE, sMAPE), and explain what situation it's designed for.

20.9 Mental Model

A time series model is a weather forecaster. It doesn't know the future. It extrapolates the patterns (trend, seasonality, momentum, mean-reversion) visible in the past. It should be honest about how much less certain it is the further out it forecasts. Stationarity is the assumption that the weather changes, but the *rules* generating the weather do not. The moment that assumption breaks, a regime shift, a new product launch, a pandemic, every classical forecasting model needs refitting. It was never modelling the cause, only the pattern. And as this chapter's own Worked Example shows, sometimes the honest, humble forecast, repeat the last value, is exactly the one that should have shipped.

Quick Reference

Scan, don't read. This appendix collects each chapter's lookup material, organised by chapter: key terms, API summaries, formulas, decision guides, and metrics. Keep it open while working; it is not meant to be read start to finish.

Chapter 1 — History & Landscape

AI / ML Timeline at a Glance

Year	Event	Why it matters
1950	Turing Test	An early operational test for machine intelligence
1956	Dartmouth workshop	Founds AI as a field (name coined in the 1955 proposal)
1986	Backpropagation popularised	Makes multi-layer networks trainable
1995	SVMs	Strong theoretical foundations for classification
1997	LSTM	Recurrent networks that learn long-range dependencies
2001	Random Forests	Still a strong tabular baseline 20+ years later
2009	ImageNet released	Large labelled dataset enables scale
2012	AlexNet	Deep learning goes mainstream
2017	Transformer ("Attention Is All You Need")	Foundation of modern LLMs
2018	BERT	Transfer learning for NLP

Year	Event	Why it matters
2020	GPT-3	Few-shot learning; scale changes the game
2022	ChatGPT	RLHF + instruction tuning → conversational AI

The Three Eras

Era	Rough dates	Core idea	Still used?
Rules & Expert Systems	1950s-1980s	Hand-coded logic	Rarely, in very narrow domains
Statistical ML	1980s-2010s	Learn patterns from labelled data	Yes — dominates tabular ML
Deep Learning / Foundation Models	2012-now	Scale neural nets on massive data	Yes — dominates unstructured data

Role Definitions

Role	Primary tool	Spends most time on
Data Analyst	SQL, BI tools	Describing what happened
Data Engineer	Spark, dbt, Kafka	Moving and storing data
Data Scientist	sklearn, PyTorch	Building predictive models
ML Engineer	MLflow, Docker, k8s	Making models work in production
Research Scientist	Papers, JAX/PyTorch	Advancing the state of the art
AI/ML Solutions Architect	All of the above	Designing systems, advising customers

Nested Scope of Terms

```
Artificial Intelligence
`-- Machine Learning
    |-- Classical ML (linear models, trees, SVMs, ensembles)
    `-- Deep Learning
        |-- CNNs (images)
        |-- RNNs / LSTMs (sequences)
        `-- Transformers (text, code, images, multimodal)
            `-- Foundation Models (pretrained, then adapted;
                |                      text, vision, multimodal)
                `-- Large Language Models (GPT, Claude, Llama, Gemini)
```

When to Use Which Approach

Situation	Reach for
Tabular data, clear features, <1M rows	XGBoost / LightGBM / Random Forest
Tabular data, need to explain decisions	Logistic Regression + SHAP
Images, audio, video	CNN or Vision Transformer
Text: classification, extraction	Fine-tuned BERT or LLM
Text: generation, chat, summarisation	Foundation model (prompt or fine-tune)
Time series forecasting	Prophet / ARIMA / Temporal Fusion Transformer
Small dataset, simple relationship	Linear / logistic regression
Need to deploy cheaply	Classical ML almost always wins

Essential pandas One-Liners

```
df.shape                          # (rows, cols)
df.dtypes                         # column types
df.isnull().sum()                 # missing values per column
df.describe()                     # summary stats (numeric)
df['col'].value_counts()          # frequency of each category
df.groupby('col')['val'].mean()   # grouped mean
df.sort_values('col', ascending=False)  # sort descending
df[df['col'] > threshold]         # filter rows
```

Essential matplotlib One-Liners

```
plt.figure(figsize=(12, 5))             # set figure size
plt.plot(x, y, label='name', color='#4E79A7')   # line
plt.scatter(x, y, s=50, alpha=0.5)     # scatter
plt.axvline(x=val, linestyle='--')     # vertical reference line
plt.xlabel('X Label')
plt.ylabel('Y Label')
plt.title('Title')
plt.legend()
plt.tight_layout()
plt.savefig('file.png', dpi=150, bbox_inches='tight')
plt.show()
```

Chapter 2 — Business Problem Framing

Key Terms

Term	Definition
Label	The exact, measurable quantity being predicted, tied to a real event
Unit of analysis	What one row of training data represents
Prediction horizon	How far in advance of the event the prediction must be made
Data leakage	A feature that encodes information unavailable at prediction time

ML Problem Type Taxonomy

Type	Label	Algorithm family	Use when
Binary classification	0 or 1	Logistic regression, tree-based, neural net	Predict yes/no event
Multi-class classification	One of N classes	Same as binary + softmax	Predict one category from several
Multi-label classification	Multiple 0/1 flags	Sigmoid per class	One item can belong to multiple categories
Regression	Continuous number	Linear regression, tree-based, neural net	Predict a quantity
Ranking	Relative order	LTR models, collaborative filtering	Order items by relevance per user
Clustering	None (unsupervised)	K-means, DBSCAN, hierarchical	Discover natural groups
Anomaly detection	Rare or unlabelled	Isolation Forest, Autoencoder, statistical	Find unusual examples
Time series forecasting	Future value	ARIMA, Prophet, TFT	Predict next N values in a sequence
Sequence labelling	Tag per token	CRF, LSTM, BERT	NER, POS tagging

Problem Statement Template

```
GOAL STATEMENT
  Purpose:      what this project is trying to do, in plain language
  Advantage:    the business benefit, tied to money or a tracked metric
  Measurement:  how success is verified, stated as a number

ML PROBLEM DEFINITION
  Label:               the exact, measurable event being predicted
  Unit of analysis:    what one training row represents
  Features available:  confirmed available at prediction time, not after
  Prediction horizon:  how far ahead of the event the prediction is made
```

```
  Is ML the right tool?  checked against the Suitability Checklist below

SUCCESS METRIC
  Business outcome:  _________________________
  Target number:     _________________________
```

A blank field is the signal Procedure 2.1 warns about: the framing isn't finished yet.

The ML Suitability Checklist

```
[ ] There is a clear, measurable prediction target
[ ] Historical data exists (or can be constructed)
[ ] Features/label relationship is stable over time
[ ] Enough examples exist (rule of thumb: >= 1,000/class)
[ ] Features at prediction time are enough to predict
[ ] A simple rule does NOT already work well enough
[ ] The cost-benefit of ML vs. rule justifies the effort
[ ] The action enabled by the prediction is operationalised
```

AWS SageMaker Quick Reference

Task	SageMaker approach	Documentation
Share a structured dataset across a team	Upload to S3, register in SageMaker Feature Store	`sagemaker.feature_store`

Chapter 3 — Maths Foundations: Statistics, Probability & Linear Algebra

Key Terms

Term	Definition
Prior	$P(A)$: belief about a hypothesis before seeing evidence
Likelihood	$P(B \mid A)$: probability of the evidence given the hypothesis

Term	Definition
Posterior	$P(A \mid B)$: updated belief after accounting for evidence
Variance	Average squared deviation from the mean, $\sigma^2 = \frac{1}{n}\sum_i (x_i - \mu)^2$
Correlation (Pearson r)	Scale-independent measure of *linear* association, $r = \mathrm{Cov}(X,Y)/(\sigma_X \sigma_Y) \in [-1, 1]$
p-value	Probability, under H_0, of data at least as extreme as what was observed
Effect size	A measure (e.g. Cohen's d) of how large a difference is, independent of sample size
Normal equations	The closed-form least-squares solution $w = (X^\top X)^{-1} X^\top y$

Probability Distributions Cheat Sheet

Distribution	Parameters	Mean	Variance	scipy / numpy	Use when
Gaussian	μ, σ	μ	σ^2	`stats.norm(mu, sigma)`	Continuous, symmetric quantities
Bernoulli	p	p	$p(1-p)$	`stats.bernoulli(p)`	Single binary outcome
Binomial	n, p	np	$np(1-p)$	`stats.binom(n, p)`	Count of successes in n trials
Poisson	λ	λ	λ	`stats.poisson(lam)`	Count of rare events in a fixed interval
Uniform	a, b	$(a+b)/2$	$(b-a)^2/12$	`np.random.uniform(a,b)`	Equal probability over a range
Exponential	λ	$1/\lambda$	$1/\lambda^2$	`stats.expon(scale=1/lam)`	Time between events

Distribution	Parameters	Mean	Variance	scipy / numpy	Use when
Beta	α, β	$\alpha/(\alpha+\beta)$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$	`stats.beta(a, b)`	Probability of a probability (Bayesian)

Bayes' Theorem

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)}, \qquad P(B) = \sum_i P(B \mid A_i)\,P(A_i)$$

In ML: B = observed features (evidence), A = the label (hypothesis). $P(A)$ is the prior (base rate); $P(B \mid A)$ is the likelihood; $P(A \mid B)$ is the posterior: the quantity a classifier is ultimately trying to produce.

Descriptive Statistics Summary

Statistic	Formula	numpy	Notes
Mean	$\bar{x} = \sum x_i/n$	`np.mean(x)`	Sensitive to outliers
Median	Middle value	`np.median(x)`	Robust to outliers
Variance	$\sigma^2 = \sum(x_i - \mu)^2/n$	`np.var(x)`	Use `ddof=1` for sample variance
Std dev	$\sigma = \sqrt{\sigma^2}$	`np.std(x)`	Same units as x
Covariance	$\sum(x_i - \mu_x)(y_i - \mu_y)/n$	`np.cov(x,y)`	Not normalised; scale-dependent. `np.cov` divides by $n-1$ by default
Correlation	$\mathrm{Cov}(X,Y)$ / $(\sigma_X\sigma_Y)$	`np.corrcoef(x,y)`	Scale-independent, $[-1, 1]$
IQR	$Q_3 - Q_1$	`np.percentile(x, [25,75])`	Robust spread measure

Common Statistical Tests

Test	Use when	Assumes	scipy call
Shapiro-Wilk	Test if data is Gaussian ($n < 5000$)	—	`stats.shapiro(x)`
t-test (independent)	Compare means of 2 groups	Gaussian, equal variance	`stats.ttest_ind(a, b)`
Welch's t-test	Compare means of 2 groups (unequal variance)	Gaussian	`stats.ttest_ind(a, b, equal_var=False)`
Mann-Whitney U	Compare 2 groups (non-parametric)	None	`stats.mannwhitneyu(a, b)`
ANOVA	Compare means of 3+ groups	Gaussian, equal variance	`stats.f_oneway(*groups)`
Chi-squared	Association between 2 categorical variables	Expected counts > 5	`stats.chi2_contingency(table)`
Pearson r	Linear correlation	Linear relationship; Gaussian only for the p-value	`stats.pearsonr(x, y)`
Spearman ρ	Monotonic correlation (non-parametric)	None	`stats.spearmanr(x, y)`

Linear Algebra Operations

Operation	Notation	numpy	Shape
Vector dot product	$\mathbf{a} \cdot \mathbf{b}$	`a @ b`	$(n,) \cdot (n,) \to$ scalar
Matrix multiply	AB	`A @ B`	$(m, n)@(n, k) \to (m, k)$
Transpose	$A^\top$	`A.T`	$(m, n) \to (n, m)$
Inverse	A^{-1}	`np.linalg.inv(A)`	$(n, n) \to (n, n)$

Operation	Notation	numpy	Shape
Solve $Ax = b$	$x = A^{-1}b$	`np.linalg.solve(A, b)`	More stable than `inv`
Eigendecomposition	$Av = \lambda v$	`np.linalg.eig(A)`	Returns (eigenvalues, eigenvectors)
SVD	$A = U\Sigma V^\top$	`np.linalg.svd(A)`	Returns (U, s, Vt)
L2 norm	$\|v\|_2$	`np.linalg.norm(v)`	$\to$ scalar
L1 norm	$\|v\|_1$	`np.linalg.norm(v, ord=1)`	$\to$ scalar
Element-wise multiply	$A \odot B$	`A * B`	Same shape as A, B

Normal Equations for Linear Regression

$$\hat{y} = Xw, \qquad w = (X^\top X)^{-1} X^\top y$$

```
X_bias = np.column_stack([np.ones(len(X)), X])
w = np.linalg.solve(X_bias.T @ X_bias, X_bias.T @ y)
# w[0] = intercept, w[1:] = feature weights
```

Limitations: $O(M^3)$ complexity in the number of features M, slow for large M. Fails if $X^\top X$ is singular (collinear features). For collinear features, use Ridge (Chapter 18) or a least-squares solver such as `np.linalg.lstsq`. For very large M, use gradient descent (Chapter 4).

Correlation Interpretation Guide

$\|r\|$ range	Strength
0.9 – 1.0	Very strong
0.7 – 0.9	Strong
0.5 – 0.7	Moderate
0.3 – 0.5	Weak
0.0 – 0.3	Negligible

Caveats: Pearson r measures *linear* correlation only; $r = 0$ does not imply independence (could be non-linear); correlation is not causation; Spearman ρ is more robust to outliers and handles monotonic non-linear relationships.

AWS SageMaker Quick Reference

Task	SageMaker approach	Documentation
Statistical analysis via UI, no code	SageMaker Data Wrangler	`sagemaker.wrangler`
Statistics on datasets too large for local memory (roughly $>$ 10 GB)	SageMaker Processing Jobs (`ScriptProcessor`)	`sagemaker.processing.ScriptProcessor`

Chapter 4 — Maths Foundations: Calculus & Optimisation

Key Terms

Term	Definition
Derivative	Rate of change of a function at a point, $\frac{df}{dx}$
Gradient	Vector of partial derivatives, $\nabla f(\mathbf{w})$; points in the direction of steepest ascent
Learning rate (α)	Step size in the gradient descent update rule
Backpropagation	Repeated application of the chain rule to compute gradients through a composed (layered) function
Convex loss	A loss surface where every local minimum is a global minimum; gradient descent reaches it with a small enough learning rate
Hessian	Matrix of second derivatives; used by second-order optimisers, $M \times M$ for M parameters

Gradient Descent Update Rule

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L(\mathbf{w})$$

Gradient Descent Variants

Variant	Batch size	numpy pattern	Pros	Cons
Batch GD	All N examples	`X.T @ (X@w - y)`	Exact gradient	Slow for large N
Stochastic GD (SGD)	1 example	`X[i] @ ( X[i]@w - y[i])`	Fast; noise can escape local minima	Very noisy, hard to converge
Mini-batch GD	B examples (32-256)	`X[batch].T @ (X[batch]@w - y[batch])`	Best of both; standard in practice	Requires batch-size tuning

Common Loss Functions

Loss	Formula	Use for	Gradient
MSE	$\frac{1}{N}\sum(\hat{y}-y)^2$	Regression	$\frac{2}{N}X^\top(X\mathbf{w}-y)$
MAE	$\frac{1}{N}\sum\lvert\hat{y}-y\rvert$	Regression, outlier-robust	$\frac{1}{N}X^\top \operatorname{sign}(X\mathbf{w}-y)$
Huber	MSE if $\lvert e\rvert<\delta$, else MAE	Regression, robust	Hybrid of MSE/MAE gradients
Binary cross-entropy	$-[y\log\hat{y}+(1-y)\log(1-\hat{y})]$	Binary classification	$(\hat{y}-y)$ with respect to the logit, with sigmoid output
Categorical cross-entropy	$-\sum_k y_k \log \hat{y}_k$	Multi-class classification	$(\hat{y}-y)$ with respect to the logits, with softmax output
Hinge	$\max(0, 1-y\hat{y})$	SVM	$-y$ if $y\hat{y}<1$, else 0

Optimiser Comparison

Optimiser	Update rule	Learning-rate sensitivity	Best for
SGD	$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L$	High	Simple models, full manual control
SGD + momentum	$\mathbf{v} \leftarrow \beta \mathbf{v} + (1-\beta)\nabla L$; $\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{v}$	Moderate	Deep learning with careful tuning
Adam	Adaptive per-parameter learning rate	Low	Default for neural networks and fine-tuning
AdamW	Adam + decoupled weight decay	Low	Transformer training (standard)
L-BFGS	Quasi-Newton (2nd-order)	Very low	Smooth, full-batch problems (classical ML); memory grows linearly with parameter count
RMSprop	Adaptive learning rate, no momentum	Moderate	RNNs, non-stationary objectives

Adam Algorithm (Simplified)

```
alpha, beta1, beta2, eps = 0.001, 0.9, 0.999, 1e-8
m, v = np.zeros_like(w), np.zeros_like(w)

for t in range(1, n_steps + 1):
    g = gradient(w)
    m = beta1 * m + (1 - beta1) * g
    v = beta2 * v + (1 - beta2) * g**2
    m_hat = m / (1 - beta1**t)
    v_hat = v / (1 - beta2**t)
    w = w - alpha * m_hat / (np.sqrt(v_hat) + eps)
```

Learning Rate Schedule Types

Schedule	Behaviour	When to use
Constant	Fixed throughout training	Simple baseline
Step decay	Halve every N epochs	Traditional CNN training
Cosine decay	Smooth decay to 0	General neural network training
Warm-up + cosine	Linear ramp then cosine decay	Transformer training (standard)
Reduce on plateau	Reduce when validation loss stalls	When unsure of a fixed schedule
Cyclic	Oscillates between min/max	Helps explore the loss landscape

$$\alpha_t = \alpha_0 \cdot \tfrac{1}{2}\left(1 + \cos\left(\frac{\pi t}{T}\right)\right), \qquad t = \text{current epoch},\ T = \text{total epochs}$$

Key Calculus Facts for ML

Concept	Formula	ML context
Derivative of x^2	$2x$	MSE gradient
Derivative of $\log(x)$	$1/x$	Cross-entropy gradient
Derivative of $\sigma(x)$	$\sigma(x)(1-\sigma(x))$	Sigmoid layer backprop
Derivative of $\text{relu}(x)$	1 if $x > 0$ else 0	ReLU layer backprop
Chain rule	$\frac{d}{dx} f(g(x)) = f'(g(x))\, g'(x)$	Backpropagation
Product rule	$\frac{d}{dx}[f \cdot g] = f'g + fg'$	Occasionally needed

scipy.optimize Cheat Sheet

```
import scipy.optimize as opt

# Minimise scalar fn
result = opt.minimize(fun, x0, method='L-BFGS-B', jac=gradient_fn)
# Find f(x) = 0
result = opt.root(fun, x0)
# 1D, no gradient needed
result = opt.minimize_scalar(fun, bounds=(a, b), method='bounded')
# Least squares fit
popt, pcov = opt.curve_fit(
    model_fn, xdata, ydata, p0=initial_params
)

result.x        # optimal parameters
result.fun      # final function value (loss)
result.success  # bool: did it converge?
result.nit      # number of iterations
```

Method	Best for	Notes
`'L-BFGS-B'`	General smooth optimisation	Usually best default
`'Nelder-Mead'`	No gradient available	Slow but robust
`'SLSQP'`	Constrained optimisation	Handles inequality constraints
`'Newton-CG'`	Smooth, exact gradient available	Fast convergence

Diagnosing Training Problems

Symptom	Likely cause	Fix
Loss goes to NaN	LR too high, or numerical overflow	Reduce LR 10x; add gradient clipping
Loss oscillates	LR too high	Reduce LR 3-10x
Loss plateaus early	LR too low, or converged to local min	Increase LR; use momentum/Adam

Symptom	Likely cause	Fix
Val loss increases while train decreases	Overfitting	Regularisation, more data
Both train and val loss plateau high	Underfitting	Larger model, more epochs, better features
Loss decreases but very slowly	LR too low, or vanishing gradients	LR schedule; batch norm; residual connections

AWS SageMaker Quick Reference

Task	SageMaker approach	Documentation
Set training hyperparameters (learning rate, epochs, batch size)	`sagemaker.estimator.Estimator(hyperparameters={...})`	`sagemaker.estimator`
Search over learning rate / hyperparameters automatically	SageMaker Automatic Model Tuning (Bayesian optimisation)	Covered in a later chapter on hyperparameter tuning

Chapter 5 — Information Theory Basics

Key Terms

Term	Definition
Self-information	$I(p) = -\log(p)$: surprise of a single event with probability p
Entropy $H(P)$	Average information content of a distribution; measures uncertainty
Cross-entropy $H(P, Q)$	Average cost of using predicted distribution Q when the truth is P
KL divergence $\mathrm{KL}(P\|Q)$	Directed "distance" of Q from P; always ≥ 0, asymmetric

Term	Definition
Mutual information $I(X;Y)$	How much knowing X reduces uncertainty about Y (and vice versa); zero iff independent
Nat vs. bit	Unit of information: natural log gives nats, $\log_2$ gives bits; 1 nat ≈ 1.4427 bits

Key Formulas

Name	Formula	When you use it
Self-information	$I(p) = -\log(p)$	Building block for everything below
Entropy	$H(P) = -\sum_x P(x) \log P(x)$	Node impurity (trees), dataset uncertainty
Cross-entropy	$H(P, Q) = -\sum_x P(x) \log Q(x)$	Classification loss function
Binary cross-entropy	$L = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$	Binary classifier loss
KL divergence	$\mathrm{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$	VAE regularisation, RLHF penalty, drift detection
Identity	$H(P, Q) = H(P) + \mathrm{KL}(P\|Q)$	Explains why minimising cross-entropy minimises KL
Mutual information	$I(X;Y) = H(X) - H(X \mid Y) = H(Y) - H(Y \mid X)$	Feature selection, representation learning

API / Function Summary

`scipy.stats.entropy(pk, qk=None, base=None, axis=0)`: returns entropy of `pk`, or $\mathrm{KL}(pk\|qk)$ if `qk` is given. `pk` need not be pre-normalised. `base=None` gives nats; `base=2` gives bits.

```
h = scipy.stats.entropy([0.5, 0.5], base=2)   # 1.0 bit
kl = scipy.stats.entropy(p, q, base=2)         # KL(p || q) in bits
```

`log_loss`, called as `sklearn.metrics.log_loss(y_true, y_pred, labels=None, normalize=True)`, returns the cross-entropy (log) loss. `y_pred` is predicted probabilities, shape `(n_samples,)` or `(n_samples, n_classes)`.

```
loss = log_loss(y_true, y_pred_proba)
```

`sklearn.feature_selection.mutual_info_classif(X, y, discrete_features='auto', n_neighbors=3, random_state=None)` (and `mutual_info_regression` for continuous targets): returns an **`ndarray`** of shape `(n_features,)`, estimated mutual information with the target in nats, non-negative.

```
mi = mutual_info_classif(X, y, random_state=42)
ranking = (
    pd.Series(mi, index=X.columns).sort_values(ascending=False)
)
```

Algorithm / Approach Comparison

Approach	Best for	Weakness	Function
Correlation (Pearson)	Quick linear-relationship screening	Misses non-linear and non-monotonic relationships entirely	`df.corr()`, `np.corrcoef`
Mutual information	Any relationship, linear or not	Needs more data to estimate reliably; no fixed scale for "how strong"	`mutual_info_classif` / `_regression`
Cross-entropy / log loss	Classifier training loss and evaluation	Only defined for probabilistic predictions, sensitive to miscalibration	`log_loss`, framework loss functions
KL divergence	Comparing two distributions directly (drift, VAE, RLHF)	Asymmetric, undefined if support doesn't match	`scipy.stats.entropy(p, q)`

Decision Guide

```
Do you need to rank features by predictive power?
|-- Yes
|   |-- Only linear relationships, want speed?
|   |   → correlation
|   `-- Non-linear relationships, or unsure?
|       → mutual_info_classif/regression
`-- No
    |-- Training/evaluating a classifier?
    |   → cross-entropy / log_loss
    `-- Comparing two known distributions
        (drift, prior vs. posterior)? → KL divergence
```

Metrics at a Glance

Metric	Range	Higher/lower is "better"?	Use when
Entropy $H(P)$	0 to $\log(K)$	Context-dependent (not a "quality" metric)	Measuring how uncertain a distribution is
Cross-entropy $H(P, Q)$	$H(P)$ to ∞	Lower is better	Classifier training loss
KL divergence	0 to ∞	Lower (closer to 0) means more similar	Distribution comparison, drift detection
Mutual information	0 to the smaller of $H(X)$ and $H(Y)$ (discrete variables)	Higher means more informative	Feature selection ranking

Numerical Pitfalls Cheat Sheet

Pitfall	Symptom	Fix
$\log(0)$	`nan` or `-inf` loss	Clip probabilities to $[\epsilon, 1-\epsilon]$, e.g. $\epsilon = 10^{-12}$

Pitfall	Symptom	Fix
Unnormalised distribution	Entropy/KL values don't match expectations	Divide by `.sum()` before computing, or let `scipy.stats.entropy` normalise for you
q has a zero where p doesn't	$\mathrm{KL}(P\|Q) = \infty$	Smooth q with a small epsilon, or use a KL variant with support handling
Comparing MI scores across different datasets	Misleading "this feature is better" conclusions	Only rank features within the same run/dataset

AWS SageMaker Quick Reference

Task	SageMaker approach	Documentation
Mutual-information feature selection at scale	`SKLearnProcessor` running a script that calls `mutual_info_classif`	SageMaker Processing SDK
Monitoring prediction distribution drift	SageMaker Model Monitor (built-in data quality / drift checks)	SageMaker Model Monitor docs
Cross-entropy training loss on GPU	Any `Estimator` (PyTorch/TensorFlow): loss defined in your training script	SageMaker Training Jobs

Chapter 6 — Python Data Science Ecosystem

Key Terms

Term	Definition
Broadcasting	NumPy applying an operation across arrays of compatible shapes without an explicit loop

Term	Definition
Series	A 1D labelled array: a single pandas column
DataFrame	A 2D labelled table, potentially with different types per column
Index	The row-label structure that lets pandas align data across sources
Estimator	A scikit-learn object (model or transformer) that follows the `.fit()`/`.predict()`/`.transform()` API

Library Purpose Table

Library	Role	When you need it
`numpy`	Fast N-dimensional arrays, maths	Any numerical computation, linear algebra
`pandas`	Labelled tabular data	Loading, cleaning, transforming, grouping datasets
`matplotlib`	Low-level plotting	Custom charts, fine-grained control
`seaborn`	Statistical visualisation	EDA, distributions, categorical comparisons
`scipy`	Scientific computing	Statistical tests, optimisation, signal processing
`scikit-learn`	ML algorithms + tooling	Models, preprocessing, cross-validation, metrics
`jupyter`	Interactive notebooks	Exploration, prototyping, sharing results

NumPy Quick Reference

```
np.array([1, 2, 3]); np.zeros((3, 4)); np.ones((2, 3)); np.eye(4)
np.arange(start, stop, step); np.linspace(start, stop, n)
np.random.randn(n); np.random.randint(low, high, size)

a.shape; a.dtype; a.ndim; a.size

a + b, a - b, a * b, a / b        # element-wise (broadcasting)
a @ b                              # matrix multiply
a.T                                # transpose
np.sum(a, axis=0)                  # sum along axis
np.mean(a), np.std(a), np.var(a)
a[a > threshold]                   # boolean indexing
np.where(condition, x, y)          # conditional selection
a.reshape(3, 4); a.flatten(); a.ravel()
```

pandas Quick Reference

```
pd.read_csv('file.csv'); pd.read_parquet('file.parquet')
df.shape; df.dtypes; df.head(n); df.info(); df.describe()
df.isnull().sum(); df.nunique()

df['col']; df[['col_a', 'col_b']]
df.loc[label]; df.iloc[n]
df[df['col'] > value]; df.query("col > @value")

df['new'] = df['a'] + df['b']
df.rename(columns={'old': 'new'}); df.drop(columns=['col'])
df.sort_values('col', ascending=False)
df.drop_duplicates(subset=['col'])
df.fillna(value); df.dropna(subset=['col'])

df.groupby('col')['val'].mean()
df.groupby(['a', 'b'])['val'].agg(['mean', 'count', 'std'])
df.pivot_table(values='v', index='r', columns='c', aggfunc='mean')
```

matplotlib / seaborn Quick Reference

```
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(x, y, color='#4E79A7', linewidth=2, label='name')
ax.scatter(x, y, s=50, alpha=0.5, c=colours, cmap='viridis')
ax.bar(categories, values, color='steelblue', edgecolor='white')
ax.set_title('Title'); ax.set_xlabel('X'); ax.set_ylabel('Y')
ax.legend()
plt.savefig('out.png', dpi=150, bbox_inches='tight')

sns.histplot(data=df, x='col', hue='group', kde=True, bins=30)
sns.boxplot(data=df, x='cat', y='val', hue='group')
sns.barplot(data=df, x='cat', y='val', errorbar='sd')
sns.scatterplot(data=df, x='a', y='b', hue='label', alpha=0.5)
sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm', center=0)
sns.pairplot(df, hue='target')
```

scikit-learn API Pattern

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
score  = model.score(X_test, y_test)

scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)  # fit on train
X_test_s = scaler.transform(X_test)  # apply to test (no fit!)

from sklearn.pipeline import Pipeline
pipe = Pipeline([
    ('scale', StandardScaler()),
    ('model', LogisticRegression()),
])
pipe.fit(X_train, y_train)
```

Common Imports Block

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

```
from sklearn.model_selection import (
    train_test_split,
    cross_val_score,
)
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.pipeline import Pipeline

plt.rcParams['figure.figsize'] = (10, 5)
sns.set_style('whitegrid')
np.random.seed(42)
```

Troubleshooting Quick Reference

Error / symptom	Most likely cause	Fix
`ModuleNotFoundError: No module named 'numpy'`	Wrong environment active	Activate the right env, then reinstall
`ValueError: Input contains NaN`	sklearn received missing values	Clean data first (Chapter 7)
`DataConversionWarning: A column-vector y was passed`	`y` is shape `(n,1)` not `(n,)`	`y.ravel()` or `y.squeeze()`
Plot is blank / not showing	`plt.show()` before data calls	Move data calls above `plt.show()`
seaborn `hue` not working	Column has NaN values	`df.dropna(subset=['hue_col'])` first

Chapter 7 — Data Cleaning & Wrangling

Key Terms

Term	Definition
MCAR	Missing Completely At Random: missingness unrelated to any variable, observed or not

Term	Definition
MAR	Missing At Random: missingness depends on observed variables, not the missing value
MNAR	Missing Not At Random: missingness depends on the missing value. Naive imputation biases results
Winsorising	Capping values beyond a percentile threshold at that threshold, rather than removing them
IQR	Interquartile range, $Q_3 - Q_1$; used to define robust, skew-resistant outlier bounds

Data Quality Problem Types

Problem	Detection	Treatment
Missing values	`df.isnull().sum()`	Impute, drop, or flag
Outliers	IQR, z-score, domain knowledge	Cap, remove, transform, or flag
Duplicates	`df.duplicated().sum()`	`drop_duplicates()`
Wrong types	`df.dtypes`	`pd.to_numeric()`, `pd.to_datetime()`, `.astype()`
Inconsistent categories	`.value_counts()`	`.str.strip().str.lower()` + `.map()`
Encoding errors	Scan `.value_counts()` for a repeated Ã-plus-character pattern	On rows containing Ã only: `.str.encode('latin1').str.decode('utf-8')` (reverses a UTF-8-read-as-Latin-1 mismatch)
Schema violations	Rule-based checks (Procedure 7.6), not just `.dtypes`	Drop, correct, or impute, once the rule that was broken is known

Missing Value Imputation Comparison

Strategy	When to use	Risk
Drop rows	$< 5\%$ missing, MCAR	Loses data
Mean fill	Numeric, symmetric	Sensitive to outliers
Median fill	Numeric, skewed	Slightly less precise than mean for normal data
Mode fill	Categorical	Can over-represent majority class
Forward fill	Time series	Wrong if gap is long
KNN imputation	Other features correlate with the missing one	Slow on large datasets
Model-based	MAR, with strong predictors of the missing column	Added complexity; risk of leakage if fit on the full dataset

Outlier Detection Methods

Method	Formula	Best for
IQR	Outside $Q_1 - 1.5 \cdot \text{IQR}$ or $Q_3 + 1.5 \cdot \text{IQR}$	Skewed distributions, robust
Z-score	$\lvert z \rvert > 3$, $z = (x - \mu)/\sigma$	Roughly normal distributions
Percentile	Below p_1 or above p_{99}	Any distribution, simple
Domain cap	Hard limits from business rules	When the valid range is known a priori
Isolation Forest	Tree-based anomaly score	Multivariate, complex patterns (Chapter 19)

dtype Conversion Table

From	To	Method
Text (string number)	`float64`	`pd.to_numeric(col, errors='coerce')`
Text (string number)	`Int64` (nullable int)	`pd.to_numeric(col).astype('Int64')`
Text (date string)	`datetime64`	`pd.to_datetime(col, errors='coerce')`
`float64`	`int64` (after removing NaN)	`.dropna().astype(int)`
Text	`category`	`.astype('category')`
`int` (0/1)	`bool`	`.astype(bool)`
`datetime64`	Text (formatted string)	`.dt.strftime('%Y-%m-%d')`

Cleaning Checklist

```
[ ] 1. Check shape (rows x cols)
[ ] 2. Check dtypes: fix wrong types immediately
[ ] 3. Count missing values per column
[ ] 4. Count and remove exact duplicates
[ ] 5. Inspect category columns: normalise strings
[ ] 6. Detect numeric outliers (IQR or z-score)
[ ] 7. Decide imputation strategy per column
[ ] 8. Apply imputation
[ ] 9. Validate: assert on domain rules
[ ] 10. Compare before/after distributions visually
```

Troubleshooting Common Errors

Error / symptom	Most likely cause	Fix
`ValueError: Input contains NaN` (sklearn)	Missing values not imputed	Check `df.isnull().sum()`, impute first

Error / symptom	Most likely cause	Fix
Category encoding produces 20+ categories	Inconsistent string values	Normalise with `.str.lower().str.strip()` + map
KNN imputer runs very slowly	Too many columns included	Include only numeric columns relevant to the target
Duplicates reappear after a merge	Join key not unique on one side	Check key uniqueness first, or pass `validate='one_to_one'` to `merge()`
`pd.to_datetime` returns all NaT	Unexpected date format	Specify `format=` explicitly, e.g. `format='%d/%m/%Y'`

This chapter has no AWS SageMaker Quick Reference. Chapter 7's cleaning operations are pandas/scikit-learn work performed before any AWS-side step. The SageMaker Feature Store material in Appendix A's Chapter 2 entry is where a cleaned dataset like this chapter's next gets registered.

Chapter 8 — SQL for Data Science

Key Terms

Term	Definition
Common table expression (CTE)	A named subquery defined with `WITH`, used for readability, not performance
Window function	A function computed across rows related to the current row, without collapsing them the way `GROUP BY` does
Partition	The `PARTITION BY` grouping a window function restarts its calculation within, analogous to `GROUP BY` but non-collapsing
Fact table	The table of events/transactions being enriched (e.g. `orders`)

Term	Definition
Dimension table	The table of descriptive attributes joined onto a fact table (e.g. `customers`, `products`)

Window Function Syntax

```
function_name() OVER (
    PARTITION BY col1, col2   -- restart calculation for each group
    ORDER BY col3             -- row order within the partition
    ROWS BETWEEN n PRECEDING  -- optional: window frame
              AND CURRENT ROW
)
```

Window Function Quick Reference

Function	Returns	Example
`ROW_NUMBER()`	Unique sequential integer (no ties)	`ROW_NUMBER() OVER ( PARTITION BY dept ORDER BY salary DESC)`
`RANK()`	Rank with gaps on ties (1,1,3)	`RANK() OVER (ORDER BY score DESC)`
`DENSE_RANK()`	Rank without gaps (1,1,2)	`DENSE_RANK() OVER ( ORDER BY score DESC)`
`NTILE(n)`	Bucket into n groups (quartiles etc.)	`NTILE(4) OVER (ORDER BY salary)`
`LAG(col, n)`	Value of `col` n rows back	`LAG(revenue, 1) OVER ( ORDER BY date)`
`LEAD(col, n)`	Value of `col` n rows ahead	`LEAD(revenue, 1) OVER ( ORDER BY date)`
`FIRST_VALUE(col)`	First value in partition	`FIRST_VALUE(order_date) OVER (PARTITION BY customer_id ORDER BY order_date)`

Function	Returns	Example
`LAST_VALUE(col)`	Last value in partition	Requires `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`
`SUM(col) OVER (...)`	Running total (add `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` if dates can tie)	`SUM(rev) OVER (ORDER BY date)`
`AVG(col) OVER (...)`	Moving / partition average	`AVG(rev) OVER ( PARTITION BY dept)`
`COUNT(*) OVER ()`	Grand total row count	Same value on every row

Aggregation Patterns

```
-- Basic group aggregation
SELECT col, COUNT(*), SUM(val), AVG(val), MIN(val), MAX(val)
FROM table GROUP BY col ORDER BY SUM(val) DESC;

-- Filter groups with HAVING
GROUP BY col HAVING SUM(val) > 1000

-- Multiple grouping levels
GROUP BY ROLLUP(region, dept)   -- subtotals at each level

-- Conditional aggregation
SELECT
    SUM(CASE WHEN status = 'completed'
             THEN revenue ELSE 0 END) AS completed_rev,
    SUM(CASE WHEN status = 'returned'
             THEN revenue ELSE 0 END) AS returned_rev
FROM orders;
```

CTE Template

```
WITH
cte_name_1 AS ( SELECT ... ),
cte_name_2 AS ( SELECT ... FROM cte_name_1 WHERE ... ),
cte_name_3 AS (
    SELECT
        *,
        RANK() OVER (PARTITION BY ... ORDER BY ...) AS rank
    FROM cte_name_2
)
SELECT * FROM cte_name_3 WHERE rank <= 10;
```

Join Types

Join	Rows kept
`INNER JOIN`	Only rows matching in both tables
`LEFT JOIN`	All rows from left, matched rows from right (`NULL`s where no match): the dominant join in analytics
`RIGHT JOIN`	Mirror of `LEFT JOIN` (rare: reorder tables and use `LEFT JOIN` instead)
`FULL OUTER JOIN`	All rows from both tables, `NULL`s where no match on either side
`CROSS JOIN`	Every combination of left × right rows (use carefully)

```
-- Most common analytical join pattern
FROM fact_table f
LEFT JOIN dim_table d ON f.foreign_key = d.primary_key
```

Date Functions (DuckDB / ANSI SQL)

```
DATE_TRUNC('month', date_col)          -- first day of month
DATE_TRUNC('year', date_col)           -- first day of year
DATE_PART('month', date_col)           -- extract month number
DATEDIFF('day', start_date, end_date)  -- days between dates
date_col + INTERVAL '30 days'          -- add 30 days
date_col + INTERVAL '1 month'          -- add 1 month
CURRENT_DATE                           -- today
```

DuckDB vs sqlite3 vs pandas.query

Feature	DuckDB	sqlite3	pandas.query
Window functions	Full support	Yes (SQLite 3.25+)	No
CTEs	Yes	Yes	No
Multi-table JOINs	Yes	Yes	No
Reads pandas DataFrames	Direct	Via manual insert	N/A: IS pandas
Parallelism	Multi-core	Single-threaded	Partial (numpy)
Aggregation on millions of rows	Fast	Slow	Slow
Setup	`pip install duckdb`	stdlib	Part of pandas

Common Analytical Query Skeletons

```
-- Top-N per group
WITH ranked AS (
    SELECT
        *,
        RANK() OVER (
            PARTITION BY group_col ORDER BY metric DESC
        ) AS r
    FROM table
)
SELECT * FROM ranked WHERE r <= 3;
```

```sql
-- Period-over-period comparison
WITH monthly AS (
    SELECT
        DATE_TRUNC('month', date_col) AS m,
        SUM(val) AS rev
    FROM t
    GROUP BY 1
)
SELECT
    m,
    rev,
    LAG(rev) OVER (ORDER BY m) AS prev,
    (rev / LAG(rev) OVER (ORDER BY m) - 1) * 100 AS pct
FROM monthly;

-- Running total
SELECT
    date_col,
    val,
    SUM(val) OVER (
        ORDER BY date_col
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM t;

-- Deduplicate (keep latest per ID)
WITH ranked AS (
    SELECT
        *,
        ROW_NUMBER() OVER (
            PARTITION BY id ORDER BY updated_at DESC
        ) AS r
    FROM t
)
SELECT * FROM ranked WHERE r = 1;
```

SQL vs pandas Decision Guide

Use SQL when	Use pandas when
Data already lives in a database or warehouse	Data is already loaded as a DataFrame
Query joins more than two tables	The transformation is custom, specific to feature engineering

Use SQL when	Use pandas when
Aggregation runs over rows too numerous to hold in memory	Task is quick exploration of a CSV never destined for a database

AWS Athena / Redshift Quick Reference

Task	AWS approach	Notes
Query Parquet/CSV directly from S3	Amazon Athena (serverless SQL)	`CREATE EXTERNAL TABLE ... LOCATION 's3://...'`
Data warehouse queries at scale	Amazon Redshift	Window function syntax very close to DuckDB's
Prefer over local pandas + DuckDB	Once data lives in S3 and is too large for memory	Billed per byte of data scanned

Chapter 9 — Exploratory Data Analysis & Visualisation

Key Terms

Term	Definition
Univariate analysis	Examining one variable in isolation (distribution, spread, outliers)
Bivariate analysis	Examining the relationship between two variables (e.g. feature vs. target)
Multivariate analysis	Examining relationships among three or more variables at once
Skewness	Asymmetry of a distribution; positive = long right tail, negative = long left tail
Kurtosis	"Tailedness" of a distribution; high kurtosis = more extreme outliers than a normal distribution
Cardinality	Number of unique values in a categorical column

Term	Definition
Data snooping	Drawing conclusions from EDA that implicitly overfit to the full dataset, including the eventual test set

Dataframe Profiling Cheat Sheet

```
df.shape                        # (rows, columns)
df.dtypes                       # column types
df.info()                       # types + non-null counts + memory
# Numeric summary: count, mean, std, min, 25/50/75%, max
df.describe()
# Categorical summary: count, unique, top, freq
df.describe(include='object')
df.isnull().sum()               # missing count per column
df.duplicated().sum()           # duplicate row count
df.nunique()                    # unique value count per column
# Random rows (better than .head() for spotting surprises)
df.sample(5)
df.select_dtypes(include='number')  # numeric columns only
df.select_dtypes(include='object')  # categorical columns only
```

Plot Type Decision Guide

```
What are you looking at?
|-- One numeric variable
|   |-- Shape/skew/modality?
|   |   -> histogram + KDE (sns.histplot(kde=True))
|   |-- Quartiles/outliers only? -> boxplot
|   `-- Shape AND quartiles? -> violin plot
|-- One categorical variable
|   `-- Frequency of each level?
|       -> bar chart (value_counts().plot(kind='bar'))
|-- Two numeric variables
|   |-- Relationship shape? -> scatter plot
|   `-- Strength across many pairs?
|       -> correlation heatmap (sns.heatmap(df.corr()))
|-- One numeric + one categorical
|   `-- Distribution shifts by group?
|       -> boxplot/violin split by category, or KDE
|-- Two categorical variables
```

```
|   `-- Association?
|       -> cross-tab + stacked bar, or pivot heatmap
`-- 3+ numeric variables
    `-- Full pairwise structure?
        -> pairplot (sns.pairplot, hue=target)
```

Plot Type Comparison

Plot	Shows	Best for	Weakness
Histogram	Frequency by value bucket	Distribution shape, skew, modality	Bin width choice changes the story
KDE plot	Smoothed density estimate	Comparing shapes of 2+ groups	Can smooth away real bimodality if bandwidth too wide
Boxplot	Median, quartiles, outliers	Quick outlier/spread comparison across many groups	Hides bimodality entirely
Violin plot	Boxplot + KDE combined	Comparing full shape across groups	Misleading with very small groups (n < ~20)
Scatter plot	Raw relationship between 2 numeric vars	Spotting non-linear relationships, clusters	Overplotting with large N (use `alpha=` or sample)
Correlation heatmap	Pearson/Spearman r for all pairs	Scanning many features for redundancy at once	Linear (Pearson) misses non-linear relationships
Pairplot	Scatter + histogram grid for all pairs	Full-dataset overview in one figure	Slow / unreadable beyond ~6 columns
Bar chart	Count or aggregate by category	Categorical frequency, group comparisons	Misleading if categories aren't sorted or truncated

Key Summary Statistics

Statistic	Formula / call	What it tells you
Mean	`series.mean()`	Central tendency, sensitive to outliers
Median	`series.median()`	Central tendency, robust to outliers
Mode	`series.mode()`	Most frequent value (numeric or categorical)
Std. deviation	`series.std()`	Typical spread around the mean
IQR	$Q_3 - Q_1$	Robust spread (middle 50% of data)
Skewness	`series.skew()`	> 0 right-skewed, < 0 left-skewed, ≈ 0 symmetric
Kurtosis	`series.kurt()`	> 0 heavier tails than normal, < 0 lighter
Pearson r	`series_a.corr(series_b)`	Linear relationship strength, -1 to 1
Spearman r	`series_a.corr(series_b, method='spearman')`	Monotonic (linear or not) relationship strength

The IQR-rule outlier bounds are:

$$[\,Q_1 - k \cdot \text{IQR},\ Q_3 + k \cdot \text{IQR}\,], \quad k = 1.5 \text{ (typical)}$$

The z-score outlier rule flags any value where:

$$\left|\frac{x - \bar{x}}{\sigma}\right| > 3$$

Here σ is the population standard deviation (`ddof=0`), which is what `scipy.stats.zscore` uses. With n values, no $|z|$ can exceed $\sqrt{n-1}$, so the rule can never fire when $n \leq 10$.

Outlier Detection Methods

Method	Assumes	sklearn / scipy
IQR rule	Nothing about distribution shape	Manual (pandas)
Z-score	Roughly normal distribution	`scipy.stats.zscore`
Isolation Forest	Multivariate anomalies	`sklearn.ensemble.IsolationForest`
Local Outlier Factor	Local density variation matters	`sklearn.neighbors.LocalOutlierFactor`

Common Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats

pd.set_option('display.max_columns', 50)
sns.set_style('whitegrid')
plt.rcParams['figure.figsize'] = (10, 5)
```

AWS SageMaker Quick Reference

Task	SageMaker approach	Notes
Automated data quality report	SageMaker Data Wrangler -> "Data Quality and Insights Report"	Flags missing values, imbalance, target leakage, outliers automatically
Interactive EDA at scale	SageMaker Studio notebook + pandas/seaborn on data from S3/Athena	Same code as local, larger instance for bigger data
Distributed profiling for very large datasets	SageMaker Processing job running a Spark/Deequ profiling script	Needed once data exceeds single-notebook memory

Task	SageMaker approach	Notes
Feature-target correlation / leakage detection	Data Wrangler "Quick Model" analysis	Trains a throwaway model to estimate feature importance during EDA

Chapter 10 — The Data Science Lifecycle

Key Terms

Term	Definition
CRISP-DM	Cross-Industry Standard Process for Data Mining: business understanding, data understanding, data preparation, modelling, evaluation, deployment
TDSP	Microsoft's Team Data Science Process: adds explicit deployment, monitoring, and governance emphasis CRISP-DM treats lightly
Project charter	A lightweight, whole-project companion to the Chapter 2 problem statement, naming stage owners and kill criteria
Pre-mortem	A structured exercise assuming the project already failed, to surface risks while still cheap to fix
Kill criteria	The explicit condition under which a project is cancelled rather than continued on sunk cost
Loop-back	A lifecycle transition to an *earlier* stage, triggered by what a later stage reveals: the norm, not the exception

The Eight Stages, Mapped to This Book

Stage	What happens	Where it lives
1. Problem framing	Translate a business ask into a precise ML problem statement	Chapter 2
2. Data collection & understanding	Identify sources, pull data, understand structure/quality/limitations	Chapters 7-9
3. Data preparation	Clean, transform, engineer features into model-ready form	Chapters 7, 14
4. Modelling	Select algorithm family, train candidates, tune hyperparameters	Chapters 11-20
5. Evaluation	Measure against the stage-1 success metric: business metric, not just accuracy	Chapters 13, 16
6. Deployment	Package the model and serve predictions: batch or real-time	Book 2, Chapter 23
7. Monitoring	Watch for data drift, model drift, operational failures	Book 2, Chapter 25
8. Iteration	Feed evaluation/production learnings back into a revised problem statement	Back to stage 1

Framework Comparison

Stage (this curriculum)	CRISP-DM	Microsoft TDSP
1. Problem framing	Business Understanding	Business Understanding
2. Data collection & understanding	Data Understanding	Data Acquisition & Understanding

Stage (this curriculum)	CRISP-DM	Microsoft TDSP
3. Data preparation	Data Preparation	Data Acquisition & Understanding (cleaning); Modeling (feature engineering)
4. Modelling	Modeling	Modeling
5. Evaluation	Evaluation	Modeling (evaluation sub-phase)
6. Deployment	Deployment	Deployment
7. Monitoring	*(not explicit)*	Deployment (ongoing) / Customer Acceptance
8. Iteration	*(implicit feedback loop)*	Explicit: loops back to Business Understanding

Key difference: CRISP-DM (1996, data-mining era) treats deployment lightly and has no explicit monitoring phase. TDSP (2016, cloud/MLOps era) treats deployment, monitoring, and governance as first-class stages. This book follows TDSP's operational emphasis.

Stage Checklist: Inputs, Outputs, Exit Criteria

Stage	Inputs	Outputs	Exit criteria
1. Problem framing	Stakeholder ask	Filled problem statement	Label, features, horizon, success metric signed off
2. Data understanding	Raw data sources	EDA report	Data quality, coverage, label reliability documented
3. Data preparation	Raw/understood data	Model-ready dataset	No known leakage; split defined; pipeline reproducible
4. Modelling	Model-ready dataset	Trained candidate model(s)	At least one model beats the baseline

Stage	Inputs	Outputs	Exit criteria
5. Evaluation	Trained model(s), held-out test set	Evaluation report	Model metric AND business metric both meet stage-1 thresholds
6. Deployment	Approved model	Live serving endpoint or batch job	Model reachable in production; rollback plan exists
7. Monitoring	Live system	Drift/performance dashboards + alerts	Named owner receives alerts; retraining trigger defined
8. Iteration	Monitoring signals, retrospective	Revised problem statement	Feeds back into stage 1

Common Failure Modes by Stage

Stage	Failure mode	Early warning sign
Problem framing	No kill criteria; ML used where a rule would do	"We'll figure out success metrics later"
Data understanding	Label doesn't reflect current business process	Label definition changed within the historical window
Data preparation	Leakage; feature not available at prediction time	Suspiciously high correlation with label (> 0.9)
Modelling	Optimising the wrong metric	Model metric improves but no business change follows
Evaluation	One-off evaluation, never repeated	No scheduled re-evaluation cadence
Deployment	Never leaves the notebook	No named deployment owner in the charter
Monitoring	No one owns production health	Alerts routed to no one, or an unread inbox
Iteration	"Done" treated as permanent	No retrospective run; no revised problem statement

Decision Guide: Should This Be a Full Lifecycle Project?

```
Does a simple rule already meet the required performance?
|-- Yes → Ship the rule. Not a lifecycle project.
`-- No
    |-- One-off analysis, no deployed system?
    |   |-- Yes → EDA/analytics task, not
    |   |        a full lifecycle project.
    |   `-- No
    |       |-- Does historical labelled data exist
    |       |   (or can it be constructed cheaply)?
    |       |   |-- No → Data collection project first.
    |       |   |        Revisit once labels exist.
    |       |   `-- Yes → Full lifecycle project.
    |       |            Use the charter template.
```

Lifecycle Stage Owners (RACI Shorthand)

Stage	Typically Responsible	Typically Accountable
Problem framing	Data scientist / SA	Business sponsor
Data understanding	Data scientist / data engineer	Data scientist
Data preparation	Data scientist / ML engineer	Data scientist
Modelling	Data scientist	Data scientist
Evaluation	Data scientist	Business sponsor (signs off on business metric)
Deployment	ML engineer / MLOps	Engineering lead
Monitoring	ML engineer / on-call	Named individual, not "the team"
Iteration	Whole team	Business sponsor

AWS SageMaker Quick Reference

Lifecycle stage	AWS service	Notes
Problem framing	*(no AWS service: human process)*	Document in the charter
Data collection & understanding	S3, Glue, Athena, SageMaker Data Wrangler	Chapters 7-9 tooling maps here
Data preparation	SageMaker Data Wrangler, SageMaker Feature Store	Chapter 14
Modelling	SageMaker Training Jobs, SageMaker Experiments	Chapters 11-20
Evaluation	SageMaker Clarify, SageMaker Model Monitor (baseline)	Chapter 16; Book 2, Chapter 21 (Clarify)
Deployment	SageMaker Endpoints (real-time), SageMaker Batch Transform	Book 2, Chapter 23
Monitoring	SageMaker Model Monitor, CloudWatch	Book 2, Chapter 25
Iteration / orchestration	SageMaker Pipelines, SageMaker Projects	Book 2, Chapter 24

Chapter 11 — Supervised Learning: Overview of Approaches

Key Terms

Term	Definition
Inductive bias	An algorithm family's built-in assumption about the shape of the relationship between inputs and outputs
Bias (statistical)	How wrong a model's assumptions are, systematically, no matter how much data it gets

Term	Definition
Variance (statistical)	How much a model's predictions change when trained on a different sample from the same population
Curse of dimensionality	As dimensionality grows, all points become roughly equidistant, so "nearest neighbour" stops being meaningful
No Free Lunch theorem	Averaged over *all possible problems*, no algorithm outperforms any other. It does not rule out algorithms that reliably win on the specific kinds of problems seen in practice
Conditional independence	Naive Bayes' assumption that features are independent of each other given the class label

Algorithm Comparison Table

Algorithm	Task	Interpretable	Non-linear	Needs scaling	Missing values	sklearn class
Logistic Regression	Classification	Yes	No (linear)	Yes	No	`LogisticRegression`
Linear Regression	Regression	Yes	No	No (yes once regularised)	No	`LinearRegression`
Ridge Regression	Regression	Yes	No	Yes	No	`Ridge`
Decision Tree	Both	Yes (if shallow)	Yes	No	Yes (sklearn 1.3+)	`DecisionTreeClassifier` / `Regressor`
K-Nearest Neighbours	Both	No	Yes	**Yes (critical)**	No	`KNeighborsClassifier` / `Regressor`
Naive Bayes	Classification	Partial	No	No	No	`GaussianNB`, `MultinomialNB`

Algorithm	Task	Inter-pretable	Non-linear	Needs scaling	Missing values	sklearn class
Random Forest	Both	No	Yes	No	Yes (sklearn 1.4+)	`RandomForestClassifier` / `Regressor`
SVM	Both	No	Yes (with kernel)	**Yes (critical)**	No	`SVC`, `SVR`
XGBoost	Both	No	Yes	No	**Yes (native)**	`xgboost.XGBClassifier`

The Universal sklearn API

```
model.fit(X_train, y_train)   # train
# Class labels (classification) or values (regression)
model.predict(X_test)
# Class probabilities (classification only)
model.predict_proba(X_test)
model.score(X_test, y_test)   # accuracy (clf) or R² (reg)

# Pipeline pattern (always use this)
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model', YourClassifier())
])
pipe.fit(X_train, y_train)
pipe.predict(X_test)
```

Bias-Variance Quick Reference

Symptom	Diagnosis	Fix
Train accuracy high, test accuracy low	High variance (overfitting)	More data, regularisation, simpler model, ensemble
Both train and test accuracy low	High bias (underfitting)	More complex model, better features, less regularisation

Symptom	Diagnosis	Fix
Large gap between CV folds	High variance	Reduce model complexity, more training data

Algorithm Selection Decision Tree

```
What is your target?
|-- Discrete category → Classification
|   |-- Need interpretability?
|   |   |-- Yes → LogisticRegression (linear) or
|   |   |         DecisionTree (max_depth=4)
|   |   `-- No  → RandomForest or XGBoost
|   |             (usually best on tabular)
|   `-- Text data? → MultinomialNB or fine-tuned
|       transformer
`-- Continuous number → Regression
    |-- Linear relationship? → LinearRegression / Ridge
    |-- Nonlinear? → RandomForestRegressor or XGBRegressor
    `-- Time series? → Specialised forecasting models
```

Key Hyperparameters to Know

Class	Key hyperparams	Notes
`LogisticRegression`	`C` (inverse regularisation, default 1.0), `max_iter`	Larger C = less regularisation
`DecisionTreeClassifier`	`max_depth`, `min_samples_leaf`, `min_samples_split`	Limit depth to prevent overfitting
`KNeighborsClassifier`	`n_neighbors` (default 5), `metric`, `algorithm`	More neighbours = smoother boundary
`RandomForestClassifier`	`n_estimators`, `max_depth`, `max_features`	Start with `n_estimators=100`
`GaussianNB`	`var_smoothing`	Rarely needs tuning

Common Imports

```
from sklearn.linear_model import (
    LogisticRegression,
    LinearRegression,
    Ridge,
)
from sklearn.tree import (
    DecisionTreeClassifier,
    DecisionTreeRegressor,
)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.ensemble import (
    RandomForestClassifier,
    RandomForestRegressor,
)
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import (
    train_test_split,
    cross_val_score,
)
from sklearn.metrics import (accuracy_score, classification_report,
                             confusion_matrix, f1_score,
                             mean_squared_error, r2_score)
```

AWS SageMaker Quick Reference

Task	SageMaker approach	Notes
Linear/logistic regression at scale	SageMaker Linear Learner built-in algorithm	Distributed training for large tabular datasets
Tree ensembles at scale	SageMaker built-in XGBoost algorithm	Managed, spot-instance-friendly training jobs
k-NN at scale	SageMaker k-NN built-in algorithm	Approximate nearest-neighbour index for large datasets

Task	SageMaker approach	Notes
Comparing algorithm families quickly	SageMaker Autopilot	Runs several candidate algorithms and reports a leaderboard, mirroring Procedure 11.2

Chapter 12 — Ensemble Methods

Ensemble Method Comparison

Method	Strategy	Reduces	Best for	Library
Bagging	Bootstrap samples, average	Variance	High-variance base learners	`Bagging Classifier`
Random Forest	Bagging + random features	Variance	Tabular data, robust baseline	`RandomForest Classifier/ Regressor`
sklearn Gradient Boosting	Sequential residual fitting	Bias + Variance	General tabular, slower	`Gradient Boosting Classifier/ Regressor`
XGBoost	GB + regularisation, pruning, sparsity	Bias + Variance	Most tabular tasks	`xgboost.XGB Classifier/ Regressor`
LightGBM	Leaf-wise GB, GOSS, EFB	Bias + Variance	Large datasets, speed-critical	`lightgbm.LGBM Classifier/ Regressor`
Stacking	Meta-learner on base predictions	Both	Squeezing last % accuracy	`Stacking Classifier/ Regressor`

Key Hyperparameters

Random Forest: `n_estimators` (100, range 100-500), `max_depth` (None), `max_features` (`'sqrt'`), `min_samples_leaf` (1, range 1-20), `n_jobs=-1`.

XGBoost: `n_estimators` (100, range 100-1000), `learning_rate`/`eta` (0.3, range 0.01-0.3), `max_depth` (6, range 3-8), `subsample` (1.0, range 0.6-1.0), `reg_alpha`/`reg_lambda` (L1/L2).

LightGBM: `n_estimators` (100, range 100-2000), `learning_rate` (0.1, range 0.01-0.1), `num_leaves` (31, range 20-150, primary complexity control), `min_child_samples` (20, range 10-100).

Feature Importance Types

Type	How computed	Strengths	Weaknesses
Impurity (Gini/variance)	Sum of impurity reduction at splits	Fast, no extra data	Biased towards high-cardinality features
Permutation	Drop in score when feature shuffled	No high-cardinality bias	Slower, needs held-out data; misleads on correlated features
SHAP	Shapley values	Per-prediction explanations	Requires `shap` library

Common Imports

```
from sklearn.ensemble import (
    RandomForestRegressor,
    GradientBoostingRegressor,
    StackingRegressor,
)
import xgboost as xgb
import lightgbm as lgb
from sklearn.inspection import permutation_importance
```

AWS SageMaker Quick Reference

Task	SageMaker approach
Train XGBoost without a custom container	Built-in `xgboost` algorithm image
Distributed training on large tabular data	Built-in algorithm + multi-instance training job

Chapter 13 — Cross-Validation & Model Selection

CV Strategy Selection Guide

Situation	Strategy	sklearn class
Standard classification	Stratified k-fold (k=5)	`StratifiedKFold`
Standard regression	k-Fold (k=5)	`KFold`
Time-ordered data	Time-series split	`TimeSeriesSplit`
Tiny dataset (<50 samples)	Leave-One-Out	`LeaveOneOut`
Hyperparam tuning + evaluation	Nested CV	`GridSearchCV` inside `cross_val_score`
Groups (same subject in train+test)	Group k-fold	`GroupKFold`

Common CV Pitfalls

Pitfall	Consequence	Fix
Preprocessing fitted on whole data, not per-fold	Data leakage: overoptimistic scores	Always use `Pipeline`
Reporting test score after using it for selection	Overoptimistic: selection bias	Nested CV or a held-out final test set
Shuffled CV on time-series data	Leaks future into training	`TimeSeriesSplit`
Not stratifying for imbalanced classes	Unstable fold scores	`StratifiedKFold`

Scoring Strings

Scoring string	Metric	Task
`'accuracy'`	Fraction correct	Classification
`'f1_weighted'`	Weighted F1	Multiclass
`'roc_auc'`	ROC-AUC	Binary classification
`'r2'`	R^2	Regression
`'neg_root_mean_squared_error'`	$-$RMSE	Regression

All regression scorers are negated, since sklearn maximises. Negate the result to get the real metric.

How Many Folds?

Dataset size	Recommendation
$<$ 50 samples	`LeaveOneOut`
50-500 samples	`k=10`
500-10,000 samples	`k=5` (default)
$>$ 10,000 samples	`k=3` or single split

Pipeline + CV Pattern (Always Use This)

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import (
    cross_val_score,
    StratifiedKFold,
)

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model', YourModel()),
])
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

```
scores = cross_val_score(
    pipe, X, y, cv=cv, scoring='accuracy', n_jobs=-1
)
```

Common Imports

```
from sklearn.model_selection import (
    train_test_split,
    KFold,
    StratifiedKFold,
    TimeSeriesSplit,
    LeaveOneOut,
    cross_val_score,
    cross_val_predict,
    cross_validate,
    GridSearchCV,
    RandomizedSearchCV,
)
```

Chapter 14 — Feature Engineering

Key Terms

Term	Definition
Feature	A measurable property of an observation used as model input
Encoding	Converting a categorical variable into numbers
Cardinality	Number of unique values in a categorical column
Leakage	A feature (or its fitted statistics) using information the model shouldn't have at prediction time
Cyclical encoding	sin/cos transform making a wrapping variable (hour, day of week) numerically continuous

Term	Definition
Interaction feature	A feature built from combining two or more other features

Encoding Strategy Comparison

Encoding	Cardinality fit	Leakage risk	Model type
One-hot	Low (< 20)	None	Any, especially linear/distance-based
Ordinal	Low, ordered	None	Any
Integer codes (`OrdinalEncoder`)	Any	None	Tree-based only
Frequency	High	None	Any
Target / mean (CV)	High	High: must use cross-fitting	Any

Scaler Comparison

Scaler	Formula	Sensitive to outliers?	Use when
`StandardScaler`	$(x - \mu)/\sigma$	Yes	Default, roughly normal data
`MinMaxScaler`	$(x - \min)$ / $(\max - \min)$	Very	Bounded input needed
`RobustScaler`	$(x - \text{median})/\text{IQR}$	No	Data has significant outliers
None	—	—	Tree-based models

Decision Guide

```
Is the column categorical?
|-- Yes
|   |-- Cardinality < 20?
|   |   |-- Has natural order? → OrdinalEncoder
|   |   `-- No order           → OneHotEncoder
|   `-- Cardinality >= 20?
|       |-- Tree-based model? → OrdinalEncoder
|       `-- Linear/other model?
|           → Frequency or CV target encoding
`-- No (numeric)
    |-- Datetime?
    |   → extract hour/dow/month + cyclical sin/cos
    |-- Right-skewed? → log1p transform
    `-- Feeding linear/distance model?
        → StandardScaler / RobustScaler
```

Common Imports

```
from sklearn.preprocessing import (
    OneHotEncoder,
    OrdinalEncoder,
    StandardScaler,
    PolynomialFeatures,
    KBinsDiscretizer,
)
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
```

AWS SageMaker Quick Reference

Task	SageMaker approach
No-code feature transforms	SageMaker Data Wrangler flow
Shared feature store (train + serve)	SageMaker Feature Store
Embed preprocessing in inference	`sklearn.Pipeline` saved via `joblib`, deployed as one model

Chapter 15 — Feature Selection & Dimensionality Reduction

Key Terms

Term	Definition
Feature selection	Choosing a subset of existing features; the rest are discarded, survivors keep their original meaning
Feature extraction	Constructing a smaller number of new features as combinations of the originals (e.g. PCA components)
Curse of dimensionality	The set of problems (sparse data, meaningless distances, overfitting) that worsen as feature count grows
Explained variance ratio	Fraction of total dataset variance captured by a given principal component
Loading	The weight of an original feature in a principal component's linear combination
Perplexity (t-SNE)	Roughly, the effective number of nearest neighbours considered per point; controls local vs. global balance
Embedding	The low-dimensional representation produced by PCA/t-SNE/UMAP

Selection Method Comparison

Method	Family	Model-aware?	Handles non-linear relationships?	Relative cost
Variance threshold	Filter	No	N/A (variance only)	Very low
Correlation with target	Filter	No	No (linear only)	Very low
Mutual information	Filter	No	Yes	Low-medium

Method	Family	Model-aware?	Handles non-linear relationships?	Relative cost
RFE / RFECV	Wrapper	Yes	Depends on base estimator	High (retrains repeatedly)
L1 / Lasso coefficients	Embedded	Yes	No (linear model)	Low (one fit)
Tree feature importance	Embedded	Yes	Yes	Low-medium (one fit)

PCA vs. t-SNE vs. UMAP

Property	PCA	t-SNE	UMAP
Linear or non-linear	Linear	Non-linear	Non-linear
Preserves	Global variance	Local neighbourhood structure	Local + more global structure than t-SNE
Deterministic	Yes	No (stochastic)	No (stochastic, but more stable)
`.transform()` on new data	Yes	No (classic implementation)	Yes (with caveats)
Interpretable components	Somewhat (loadings)	No	No
Typical use	Preprocessing for modelling, or visualisation	Visualisation only	Visualisation, sometimes clustering preprocessing
Relative speed on large data	Fast	Slow	Faster than t-SNE

Decision Guide

```
Need the output to feed a downstream model (not a plot)?
|-- Yes
|   |-- Want to keep original feature meaning?
|   |   → Feature selection (filter/wrapper/embedded)
|   `-- OK losing interpretability for compactness?
|       → PCA
`-- No - just exploring / visualising
    |-- Fast look, large dataset (100k+ rows)? → UMAP
    |-- Best local cluster separation, small-medium
    |   dataset? → t-SNE
    `-- Fast, deterministic, reproducible 2D view? → PCA
```

API / Function Summary

```python
# Features with variance <= threshold are removed
VarianceThreshold(threshold=0.0)

SelectKBest(
    # or f_classif, chi2, mutual_info_regression, f_regression
    score_func=mutual_info_classif,
    k=10,
)

mutual_info_classif(
    X, y, discrete_features='auto', random_state=42
)

RFE(estimator, n_features_to_select=10, step=1)
# .support_, .ranking_, .n_features_
RFECV(estimator, step=1, cv=5, scoring='accuracy')

SelectFromModel(estimator, prefit=False, threshold='median')

PCA(
    # int, float in (0,1) (variance target), or None (all)
    n_components=None,
    random_state=42,
    svd_solver='auto',
)
# .explained_variance_ratio_, .components_ (loadings),
# .n_components_
```

```
# Like PCA but for sparse matrices, no centering
TruncatedSVD(n_components=10, random_state=42)

# PCA for data with non-linear structure; the kernel trick
# from Chapter 11's SVM section, applied to PCA
KernelPCA(n_components=2, kernel='rbf', gamma=None, random_state=42)
# No simple inverse back to the original feature space

TSNE(
    n_components=2,
    perplexity=30,
    learning_rate='auto',
    random_state=42,
    max_iter=1000,
)
# No .transform() for new points in the classic implementation

UMAP(n_components=2, n_neighbors=15, min_dist=0.1, random_state=42)
# supports .transform() on new data, unlike classic t-SNE
```

Common Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import (
    VarianceThreshold,
    SelectKBest,
    mutual_info_classif,
    mutual_info_regression,
    RFE,
    RFECV,
    SelectFromModel,
)
from sklearn.decomposition import PCA, TruncatedSVD, KernelPCA
from sklearn.manifold import TSNE
from sklearn.ensemble import RandomForestClassifier
# pip install umap-learn
import umap
```

AWS SageMaker Quick Reference

Task	SageMaker approach	Documentation
Distributed PCA on very large datasets	Built-in `PCA` algorithm	`sagemaker.PCA` estimator
No-code feature importance / correlation analysis	SageMaker Data Wrangler analyses	`sagemaker.wrangler.processing.DataWranglerProcessor`
Interactive exploratory dimensionality reduction	SageMaker Studio notebook with sklearn/UMAP	Standard notebook instance / Studio kernel
Feature selection as a repeatable pipeline step	SageMaker Processing job (SKLearnProcessor) wrapping the fitted selector	`sagemaker.sklearn.processing.SKLearnProcessor`

Chapter 16 — Model Evaluation

Classification Metrics — Formulas

Metric	Formula	Higher = better?
Accuracy	$(TP + TN)/(TP + FP + FN + TN)$	Yes
Precision	$TP/(TP + FP)$	Yes
Recall (Sensitivity)	$TP/(TP + FN)$	Yes
F1 Score	$2PR/(P + R)$	Yes
ROC-AUC	Area under ROC curve	Yes (0.5 = random)
PR-AUC / Avg Precision	Area under PR curve	Yes
Log Loss	$-\text{mean}(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$	No (lower better)

Regression Metrics — Formulas

Metric	Formula	Sensitive to outliers?
MSE	$\text{mean}((y - \hat{y})^2)$	Yes
RMSE	$\sqrt{\text{MSE}}$	Yes
MAE	$\text{mean}(\lvert y - \hat{y} \rvert)$	Less
MAPE	$\text{mean}(\lvert y - \hat{y} \rvert / y) \times 100$	Less
R^2	$1 - SS_{res}/SS_{tot}$	Moderate

RMSE/MAE ratio > 2 indicates outliers are significantly influencing RMSE.

Metric Selection Guide

Business context	Recommended metric
Balanced classes, symmetric errors	Accuracy or F1
Imbalanced classes	F1, PR-AUC
FP expensive (spam filter)	Precision
FN expensive (cancer, fraud)	Recall
Compare models before setting threshold	ROC-AUC
Probabilities matter (pricing, risk)	Log Loss + calibration
Rare event detection	PR-AUC (ROC-AUC is misleading)

Calibration Interpretation

Curve shape	Meaning	Fix
On the diagonal	Well-calibrated	Nothing needed
S-curve (probabilities avoid 0 and 1)	Under-confident at the extremes; common in Random Forests	Isotonic regression or Platt (sigmoid) scaling
Below diagonal	Over-estimates: observed rate is lower than predicted	Isotonic regression or Platt (sigmoid) scaling

sklearn API Quick Reference

```
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    roc_auc_score,
    average_precision_score,
    log_loss,
    confusion_matrix,
    classification_report,
    roc_curve,
    precision_recall_curve,
    mean_squared_error,
    mean_absolute_error,
    r2_score,
)
from sklearn.calibration import (
    calibration_curve,
    CalibratedClassifierCV,
)
```

AWS SageMaker Quick Reference

Task	Approach
Log metrics during training	`sagemaker.Session().log_metric(name, value, step)`
Automatic model quality monitoring	SageMaker Model Monitor: `ModelQualityMonitor`

Chapter 17 — Hyperparameter Optimisation

Search Strategy Comparison

Strategy	Pros	Cons	Use when
Manual (one-at-a-time)	Simple, interpretable	Misses interactions	Debugging only
Grid Search	Exhaustive over grid	Exponential in #params	≤ 3 params, fast model
Random Search	Wide coverage, parallelisable	No guidance from past trials	Most situations
Bayesian (Optuna)	Sample-efficient, prunes bad trials	Trials run in sequence	Expensive models

Rule of thumb: random search with 50-100 trials beats grid search on most problems, especially once a grid is restricted to just a couple of values per dimension. This chapter's Worked Example measures the gap directly and shows why: a coarse grid can only sample the corners of the search space, never the interior.

Key Hyperparameters by Algorithm

XGBoost/LightGBM: `n_estimators` (100-1000), `max_depth` (3-8), `learning_rate` (0.01-0.3, log scale), `subsample` (0.6-1.0), `reg_alpha`/`reg_lambda`.

Random Forest: `n_estimators` (100-1000), `max_depth` (5-30 or None), `max_features` (`'sqrt'`, `'log2'`), `min_samples_leaf` (1-10).

Logistic Regression: `C` (0.001-100, log scale), `penalty` (`'l1'`, `'l2'`, `'elasticnet'`).

Optuna Quick-Start

```
import optuna
optuna.logging.set_verbosity(optuna.logging.WARNING)

def objective(trial):
    params = {
        'n_estimators': trial.suggest_int(
            'n_estimators', 50, 500
```

```
        ),
        'max_depth': trial.suggest_int('max_depth', 2, 10),
        'learning_rate': trial.suggest_float(
            'learning_rate', 0.01, 0.3, log=True
        ),
    }
    model = MODEL_CLASS(**params, random_state=42)
    scores = cross_val_score(
        model, X_train, y_train, cv=5, scoring='roc_auc'
    )
    return scores.mean()

study = optuna.create_study(
    direction='maximize',
    sampler=optuna.samplers.TPESampler(seed=42),
)
study.optimize(objective, n_trials=100)
```

Optuna `suggest_*` API

```
trial.suggest_int('name', low, high)
trial.suggest_float('name', low, high)
trial.suggest_float('name', low, high, log=True)
trial.suggest_categorical('name', ['a', 'b', 'c'])
```

Validation Set Hygiene

```
Training data → hyperparameter search (CV folds)
             → pick best config
Test data     → evaluate best model ONCE, at the very end
```

Never tune on the test set. After tuning, the CV score is optimistic. The test score is the honest estimate.

AWS SageMaker HPO Quick Reference

Feature	SageMaker equivalent
Bayesian search	`strategy='Bayesian'` in `HyperparameterTuner`
Random search	`strategy='Random'`
Parallel trials	`max_parallel_jobs=N`

Chapter 18 — Regularisation

Regularisation Method Comparison

Method	Penalty	Zeroes coefficients?	Best for
Ridge (L2)	$\alpha \sum w_i^2$	No: shrinks all	Most features informative, correlated features
Lasso (L1)	$\alpha \sum \lvert w_i \rvert$	Yes: sparse solution	Many irrelevant features, automatic selection
Elastic Net	$\alpha(\rho \sum \lvert w_i \rvert + \frac{1}{2}(1-\rho) \sum w_i^2)$	Yes	Correlated features + sparsity
Dropout	Random zero-out during training	N/A	Neural networks
Early stopping	Stop when val loss stops improving (with patience)	N/A	Neural nets, gradient boosting

Bias-Variance Trade-off

	Low Complexity	High Complexity
Bias	High	Low
Variance	Low	High
Test Error	High ("underfitting")	High ("overfitting")

Learning Curve Diagnosis

Pattern	Diagnosis	Fix
Both curves converge to a similarly high error	High bias	More flexible model, better features
Train error stays low, validation error stays higher, gap persists	High variance	More data, more regularisation

Key Parameters

Model	Regularisation param	Direction
Ridge / Lasso	`alpha`	Larger = more regularisation
ElasticNet	`alpha`, `l1_ratio`	`l1_ratio`: 0=Ridge, 1=Lasso
LogisticRegression	`C`	Smaller C = more regularisation ($C = 1/\alpha$)
XGBoost/LightGBM	`reg_alpha` (L1), `reg_lambda` (L2)	Larger = more

Regularisation in Other Model Types

Model type	L2 equivalent	L1 equivalent
XGBoost/LightGBM	`reg_lambda`	`reg_alpha`
Neural networks	weight decay in optimiser	None (dropout is the main regulariser)
Decision trees	`max_depth`, `min_samples_leaf` (structural)	—

sklearn API

```
from sklearn.linear_model import (
    Ridge, RidgeCV, Lasso, LassoCV, ElasticNet, ElasticNetCV,
    LogisticRegression, LogisticRegressionCV,
)
```

Dropout Quick Reference

```
import torch.nn as nn
# Zero out 50% of activations during training
nn.Dropout(p=0.5)
# model.train() -> dropout active
# model.eval() -> dropout disabled
```

AWS SageMaker Equivalent

```
from sagemaker.linear_learner import LinearLearner
ll = LinearLearner(
    predictor_type='binary_classifier', l1=0.01, wd=0.001
)
```

Chapter 19 — Unsupervised Learning

Key Terms

Term	Definition
Clustering	Grouping similar points without ground-truth labels
Anomaly detection	Identifying points that don't fit the pattern of the rest of the data
Centroid	Mean position of all points assigned to a cluster (K-Means)
Inertia	Sum of squared distances from each point to its assigned centroid
Core point	A point with at least `min_samples` neighbours within `eps` (DBSCAN)
Contamination	Expected fraction of anomalies in a dataset
BIC / AIC	Bayesian/Akaike Information Criterion: single scores for choosing GMM component count; BIC penalises extra components more as data grows, AIC keeps the penalty fixed

Algorithm Comparison

Approach	Best for	Weakness
K-Means	Fast, large datasets, roughly spherical clusters	Must specify K; assumes equal-size, spherical clusters
Gaussian Mixture Model	Elliptical clusters, soft/probabilistic membership	Slower (full covariance per component); can still converge to a poor local optimum
DBSCAN	Arbitrary-shaped clusters, automatic outlier detection	Sensitive to `eps`/`min_samples`; struggles with varying density

Approach	Best for	Weakness
Agglomerative	Small datasets, exploratory analysis, dendrograms	At least $O(n^2)$ time and memory: impractical above ~10k rows
Isolation Forest	High-dimensional anomaly detection, scales well	Random results across runs; needs `contamination` estimate

Clustering Evaluation Metrics

Metric	Range	Higher better?	Use when
Silhouette score	-1 to 1	Yes	General-purpose cluster quality
Davies-Bouldin index	0 to ∞	No	Comparing compactness/separation across K
Calinski-Harabasz index	0 to ∞	Yes	Fast, good for large datasets
Adjusted Rand Index	-0.5 to 1	Yes	Only with ground-truth labels available

Decision Guide

```
Roughly know cluster count, likely spherical? → K-Means
Need soft membership or elliptical clusters? → Gaussian Mixture Model
Unknown K, or irregular shapes / noise? → DBSCAN
Small dataset (<10k), want to explore visually?
    → Agglomerative + dendrogram

High-dim, large anomaly-detection dataset?
    → IsolationForest
Density varies a lot across feature space?
    → LocalOutlierFactor
Small, clean dataset, tight non-linear boundary?
    → OneClassSVM
```

Common Imports

```
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.mixture import GaussianMixture
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor, NearestNeighbors
from sklearn.metrics import (
    silhouette_score,
    davies_bouldin_score,
    adjusted_rand_score,
)
```

AWS SageMaker Quick Reference

Task	SageMaker approach
Managed K-Means at scale	`sagemaker.KMeans` built-in algorithm
Managed anomaly detection	`sagemaker.RandomCutForest` built-in algorithm
Real-time anomaly scoring on streams	Managed Service for Apache Flink calling a deployed model (Kinesis Data Analytics for SQL, with `RANDOM_CUT_FOREST`, retired in January 2026)

Chapter 20 — Time Series Analysis

Key Terms

Term	Definition
Trend	The long-run direction of the series
Seasonality	A pattern that repeats at a fixed, known period
Stationarity	Statistical properties (mean, variance, autocorrelation) constant over time
Differencing	Subtracting a lagged value to remove trend or seasonality

Term	Definition
ACF / PACF	Autocorrelation / partial autocorrelation functions
Unit root	Property of a non-stationary series; the ADF test's null hypothesis

ACF / PACF → Model Order Cheat Sheet

Pattern	Suggests
ACF cuts off sharply after lag q, PACF decays gradually	MA(q)
PACF cuts off sharply after lag p, ACF decays gradually	AR(p)
Both decay gradually	Mixed ARMA(p,q): use `auto_arima`
ACF decays very slowly	Non-stationary: difference first
Spike at lag m and multiples of m	Seasonal component at period m

Forecasting Method Comparison

Approach	Best for	Weakness
Naive / seasonal naive	Baseline: always compute first	No real "modelling"
Exponential smoothing	Simple, robust, minimal tuning	Limited to trend + one seasonal pattern
ARIMA / SARIMA	Single series, built-in prediction intervals	Requires stationarity, manual/auto order selection
Lag-feature gradient boosting	Many related series, external regressors	No native uncertainty intervals

Forecast Evaluation Metrics

Metric	Formula	Use when
MAE	$\frac{1}{n}\sum\lvert y_i - \hat{y}_i\rvert$	Default; robust to outliers
RMSE	$\sqrt{\frac{1}{n}\sum(y_i - \hat{y}_i)^2}$	Penalise large errors more
MAPE	$\frac{100}{n}\sum\lvert y_i - \hat{y}_i\rvert/\lvert y_i\rvert$	Cross-scale comparison; avoid near zero
sMAPE	Bounded 0-200% variant of MAPE	More stable near zero

API Quick Reference

```
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.model_selection import TimeSeriesSplit
import pmdarima as pm

decomp = seasonal_decompose(x, model='additive', period=7)
# Reject non-stationarity if p < 0.05
adf_stat, p_value, *_ = adfuller(x, autolag='AIC')

model = SARIMAX(endog, order=(p,d,q), seasonal_order=(P,D,Q,m))
fitted = model.fit(disp=False)
forecast = fitted.get_forecast(steps=28)
```

Decision Guide

```
Stationary (ADF p < 0.05)?
|-- No → difference it, re-test
`-- Yes
    |-- Single series, need uncertainty intervals?
    |   → ARIMA/SARIMA
    |-- Many related series / external features?
    |   → Lag-feature gradient boosting
    |-- Missing data / holiday effects? → Prophet
    `-- Very large dataset, many series?
        → Deep learning (LSTM / TFT)
```

AWS SageMaker Quick Reference

Task	SageMaker approach
AutoML forecasting across many series	SageMaker Canvas time-series forecasting (Amazon Forecast, `create_auto_predictor`, for existing accounts only)
Custom deep learning forecaster	SageMaker built-in DeepAR algorithm
No-code forecasting	SageMaker Canvas

Answers to Exercises

Chapter 1 — History & Landscape

Exercise 1.1 [W]. (a) Random Forests: Statistical ML; an ensemble of decision trees learned from labelled examples, no hand-coded rules. (b) BERT: Deep Learning & Foundation Models; a pretrained transformer, part of the post-2012 scale-driven era. (c) MYCIN: Rules & Expert Systems; a hand-coded if-then medical diagnosis system from the 1970s, predating any learning from data.

Exercise 1.2 [W]. They're nested subsets, not synonyms. Artificial Intelligence is the broadest term: any system exhibiting intelligent behaviour, including hand-coded rule systems. Machine Learning is the subset of AI that learns from data rather than being explicitly programmed. Deep Learning is the subset of ML that specifically uses neural networks. Every deep learning system is machine learning and AI. But not every AI system is machine learning (expert systems aren't), and not every ML system is deep learning (Random Forests aren't).

Exercise 1.3 [A]. The answer depends on the three milestones chosen. The mechanical step: append the new rows, then run `df_milestones[df_milestones['year'] > 2017]` (Procedure 1.2) and count the resulting rows. Any milestone dated 2018 or later (BERT, GPT-3, ChatGPT, GPT-4, etc.) will count; anything from Era 1 or Era 2 will not.

Exercise 1.4 [A]. Reinforcement learning first exceeds classical ML in **2019** (2,231 papers vs. 1,967). That's four years after deep learning crossed the same line, in 2015. RL's climb follows the deep reinforcement learning results of the mid-2010s: DeepMind's DQN playing Atari games (2013, published in *Nature* in 2015) and AlphaGo (2016). In 2024, reinforcement learning outnumbers classical ML by roughly **1.9x** (5,533 vs. 2,850). That's a much smaller gap than deep learning's 4.8x over classical ML that year.

Exercise 1.5 [E]. Something like: "With 3,000 labelled examples on a tabular problem, a foundation model is very unlikely to beat a well-tuned XGBoost or LightGBM model on accuracy. It will also cost more to run and be harder to explain to a regulator or auditor. I'd suggest starting with gradient-boosted trees as a baseline this week. If that doesn't hit the target, we can talk about where a foundation model might help. But it's not the first thing to reach for here." The point is to redirect from "AI" as a default answer. The answer should be a specific technique matched to data volume, data type, and interpretability needs.

Exercise 1.6 [E]. Two options. (1) Explicitly filter or mask the first nonzero transition (or any `inf`/`NaN` value) before calling `nlargest()`, e.g. `growth.replace([np.inf, -np.inf], np.nan).dropna()`. This is the wrong choice if a genuine, very large but finite growth spike should be preserved: a blanket `inf` filter can be too aggressive if applied without checking what's being excluded. (2) Add a minimum-volume floor (e.g. only consider percentage growth once the series exceeds some absolute threshold, like 1k papers/year), so early near-zero noise never enters the ranking at all. This is the wrong choice if the point of the analysis is specifically to capture *early* inflection points before volume is large. That's exactly what this filter would discard.

Exercise 1.7 [R]. Discussion only. No single answer. Domingos' essay makes the point (among others) that "more data beats a cleverer algorithm" up to a point, and that feature engineering matters more than the choice of learning algorithm for most practical problems. Applied to Listing 1.7's finding: the 2015 crossover and the roughly 5x gap by 2024 could be read as "deep learning is a better algorithm." Domingos' framing suggests a different reading. Deep learning unlocked the ability to *use* far more data, with far less hand-engineered feature work than classical ML. That's consistent with this chapter's account of GPU compute and large labelled datasets as the real enablers, not a sudden algorithmic insight in isolation.

Chapter 2 — Business Problem Framing

Exercise 2.1 [W]. "Customer satisfaction" is not measurable or tied to a specific event. It's a survey construct, not something observable in transactional data at a given moment. A usable label ties it to something concrete and time-bound, e.g. `submitted_cancellation_request_within_30d` or `nps_score_below_6_next_survey`.

Exercise 2.2 [W]. (a) Clustering: no target variable, discovering natural groups. (b) Regression: predicting a continuous dollar quantity. (c) Binary classification: fail or not fail within a fixed 48-hour horizon. Predicting *which* failure mode is a multi-class extension.

Exercise 2.3 [A]. `days_since_last_price_increase` is safe only if the price increase is known and dated *before* the prediction point for every row. It can't be retroactively determined by something that happens after prediction time, such as a price change made *in response to* early churn signals. If price changes are scheduled independently of individual customer behaviour, the feature is safe. If the business sometimes offers reactive discounts to at-risk customers (which would correlate with the label), it is a leakage risk. It should be excluded, or replaced with a feature computed strictly from data available before the churn-risk window.

Exercise 2.4 [A]. On the class-weighted logistic regression from Listing 2.5, the three thresholds cost $9,345 (0.3: 1,869 false positives, 0 false negatives), $18,255 (0.5: 651 false positives, 75 false negatives), and $24,785 (0.7: 37 false positives, 123 false negatives). 0.3 wins, and by a wide margin. With `fp_cost=5` and `fn_cost=200`, a single missed churner

costs as much as 40 false alarms, so the cost-minimising threshold is pulled towards the low end of the swept range even though it means flagging almost every customer in the test set (this model's probabilities never drop below about 0.32, so 0.3 is close to "flag everyone"). The full 50-point sweep confirms it: the true minimum sits at a threshold of 0.345 ($9,185), just above 0.3, not at some higher, more conservative-looking value.

Exercise 2.5 [E]. A model routing internal support tickets to one of a few roughly equal-sized teams is a reasonable case for accuracy. Misrouting costs a short reassignment step, not a missed high-stakes event. Classes are close to balanced, and the cost of a false positive/negative is symmetric and low. The imbalance argument specifically targets skewed class distributions with asymmetric costs (fraud, churn, disease), where accuracy can be trivially gamed by always predicting the majority class. Symmetric-cost, balanced-class problems don't have that failure mode.

Exercise 2.6 [E]. With zero historical failure examples, the checklist fails on "Historical data exists (or can be constructed)": no failure has ever been recorded: supervised anomaly detection isn't viable yet. The right move is unsupervised/statistical anomaly detection (e.g. an Isolation Forest or simple statistical control limits on sensor readings) to flag unusual readings without labels, paired with a process to have engineers confirm or reject flagged events. That builds the labelled dataset needed for a future supervised model rather than pretending one exists today.

Exercise 2.7 [R]. Discussion only. No single answer. Consider: Wickham's iterative framing treats "define the problem" as something that is revisited as data exploration reveals what's measurable. This chapter instead presents the three steps as sequential gates completed before touching data. In practice the two aren't fully in conflict: the problem statement in Procedure 2.1 is meant to be revised, not treated as immutable. But the chapter's ordering is a stronger claim than Wickham's about how much framing must be locked down before EDA begins. Worth considering what breaks if a "final" problem statement is written before *any* data has been looked at, versus after a first exploratory pass.

Chapter 3 — Maths Foundations: Statistics, Probability & Linear Algebra

Exercise 3.1 [W]. The 99% sensitivity figure describes $P(\text{positive} \mid \text{disease})$, not $P(\text{disease} \mid \text{positive})$. Those two conditional probabilities are not the same quantity, despite sounding similar. Bayes' theorem forces the base rate into the calculation. The disease is rare (1% prevalence), so the pool of healthy people is roughly 99 times larger than the pool of diseased people. Even a modest 5% false-positive rate among that huge healthy majority produces far more false positives in absolute terms than the true positives produced by the small diseased minority testing correctly. The 99% figure describes the test's behaviour *conditioned on*

already knowing someone is sick. It says nothing about what to conclude *before* knowing that. That's exactly the direction the diagnosis question runs.

Exercise 3.2 [W]. (a) Bernoulli: a single binary outcome (spam or not) with no notion of counting across trials. (b) Poisson: counting rare, independent events (fraud) within a fixed interval (an hour), governed by a rate. (c) Binomial: the sum of 50 independent Bernoulli trials (each flip), counting successes. (d) Gaussian: measurement error from many small, independent sources of noise summing together is the textbook Central Limit Theorem scenario.

Exercise 3.3 [A]. Computing `df.corr()` restricted to `sepal width (cm)` against the other three iris features gives: `petal length (cm)` $r \approx -0.428$, `petal width (cm)` $r \approx -0.366$, and `sepal length (cm)` $r \approx -0.118$. The weakest (closest to zero) is `sepal length (cm)` at $r \approx -0.118$, negligible by the Correlation Interpretation Guide. This does match the intuition that sepal width is the odd one out. The other three iris measurements are strongly inter-correlated (petal length and petal width alone correlate at $r = 0.963$), while sepal width barely tracks any of them, including the other sepal measurement.

Exercise 3.4 [A]. Extending Listing 3.7 to `X_raw = df[['petal length (cm)', 'sepal length (cm)']].values` and solving gives weights `w ~ [-0.0090, 0.4494, -0.0822]` (intercept, petal-length coefficient, sepal-length coefficient) and $R^2 \approx 0.9290$. That compares to $R^2 \approx 0.9271$ for the single-feature model. The improvement is real but tiny, about 0.002 of additional variance explained. The new coefficient on `sepal length (cm)` is small and negative, suggesting it's picking up a small amount of residual signal (likely correlated with, rather than independent of, petal length) rather than contributing meaningful new information. Adding the second feature does not meaningfully improve the fit.

Exercise 3.5 [A]. Running Procedure 3.4 on `petal length (cm)` for `setosa` vs. `versicolor` gives $t \approx -39.49$, $p \approx 5.4 \times 10^{-62}$, and Cohen's $d \approx -7.90$. The difference is both statistically significant (the p-value is astronomically below any conventional threshold) and practically enormous. A Cohen's d near 8 is far beyond the "large" cutoff of 0.8. These two species' petal lengths barely overlap at all.

Exercise 3.6 [E]. Before agreeing to ship on "$p = 0.002$" alone: (1) What is the effect size, and is it large enough to matter for the business metric it's supposed to move? A real but tiny lift can hit $p = 0.002$ purely from a large sample size, exactly as the chapter's 0.9000→0.9001 accuracy example illustrates. (2) What is the confidence interval around the effect, and does its lower bound still represent a meaningful improvement, or does it include effect sizes too small to justify shipping? (3) How many other comparisons were run before this one was reported? Was this the only test, or one of several? If several, an uncorrected p-value overstates the evidence (multiple-comparisons / p-hacking risk).

Exercise 3.7 [E]. Take two features, `income` (values like 30000, 60000, 90000) and `age` (values like 25, 45, 65), across three data points: $P_1 = (30000, 25)$, $P_2 = (60000, 45)$, $P_3 = (90000, 65)$. The unscaled Euclidean distance between P_1 and P_2 is $\sqrt{(30000)^2 + (20)^2} \approx 30000.007$. `income` alone accounts for essentially the entire distance. `age`'s contribution

is invisible to four significant figures. After standardising both features to mean 0 and standard deviation 1 (via `(x - mean) / std`, with the population `std` (`ddof=0`) across the three points, as `StandardScaler` uses; pandas' default `.std()` uses `ddof=1` and gives ± 1 and a distance of 1.41 instead), each feature's z-scores follow the same pattern: roughly $-1.22, 0, 1.22$ for both, since both are evenly spaced 3-point sequences. The scaled Euclidean distance between P_1 and P_2 becomes $\sqrt{(1.22)^2 + (1.22)^2} \approx 1.73$, with income and age now contributing equally. The raw-scale calculation was entirely dominated by the feature with the larger numeric range, even though nothing about the problem says income should matter 1500x more than age.

Exercise 3.8 [R]. Discussion only. No single answer. Candidates worth pursuing: Singular Value Decomposition ($A = U\Sigma V^\top$) appears only in Appendix A's Quick Reference. The chapter itself never discusses it. It underlies PCA's actual numerical implementation: `sklearn.decomposition.PCA` uses SVD rather than eigendecomposing the covariance matrix directly, for numerical stability. It reappears in Chapter 15's dimensionality reduction. Alternatively, positive-definiteness and matrix rank: Murphy's Appendix B covers when $X^\top X$ is guaranteed invertible. That connects directly back to this chapter's warning that the normal equations fail under collinear features, and explains *why* that failure happens, rather than just stating that it does.

Chapter 4 — Maths Foundations: Calculus & Optimisation

Exercise 4.1 [W]. The gradient's each component $\partial f / \partial w_i$ tells you how f changes as w_i increases. By construction, moving in the direction of that vector increases f fastest. That's what "steepest ascent" means. Since the goal in training is to *decrease* the loss, the correct move is the exact opposite direction, $-\nabla f(\mathbf{w})$. That's what the gradient descent update rule ($\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla f(\mathbf{w})$) implements.

Exercise 4.2 [W]. Oscillation followed by a NaN is the "learning rate too high" pattern carried to its extreme. The steps have been overshooting the minimum by a growing margin each time: each overshoot lands further up the opposite wall of the loss surface than the last. Eventually a step produces a value large enough to overflow floating point. The first thing to try is reducing the learning rate by roughly 10x, since Listing 4.2's table specifically maps both "oscillating" and "NaN or infinity" back to the same root cause.

Exercise 4.3 [A]. At $\alpha = 1.1$, the per-step contraction factor is $(1 - 2\alpha) = 1 - 2.2 = -1.2$, whose magnitude exceeds 1. So instead of shrinking towards zero, x grows in magnitude every step and flips sign each time. Starting at $x = 3.0$, the sequence runs approximately $3.0 \to -3.6 \to 4.32 \to -5.184 \to 6.221 \to -7.465$ over the first 5 steps, diverging geometrically. This is the 1D version of the same "too large" divergence shown for linear regression in the Worked Example's Stage 3. The general rule for $f(x) = x^2$ is that gradient descent converges only for $0 < \alpha < 1$ (i.e. $|1 - 2\alpha| < 1$).

Exercise 4.4 [A]. At $\alpha = 0.5$, five times the original 0.1, the loss reaches the optimum after just **1 epoch**: `w = [-0.689864, 80.790445]`, loss `406.5247`, an exact match to the closed-form answer, and it stays there for every epoch after. That's not an approximation rounding down to "epoch 1"; it's exact. This particular dataset's Hessian, $\frac{2}{N}X^\top X$, works out to exactly $2I$: the standardised feature column is, by construction, uncorrelated with the constant bias column, so the two parameters have identical curvature and no cross-term. A learning rate of $\alpha = 0.5$ satisfies $\alpha \cdot 2 = 1$ exactly, which zeroes out $(I - \alpha H)$ and collapses gradient descent to the closed-form solution in a single step, regardless of the starting point. This is a property of this specific, well-conditioned, standardised, two-parameter problem, not a general rule for $\alpha = 0.5$. It's a sharper version of the chapter's "too small wastes epochs" point. On a well-conditioned problem, a larger, still-stable learning rate does not just converge faster. It can hit the exact optimum directly. 0.5 remains well inside the stable range for this loss surface, short of the divergence seen at 1.5.

Exercise 4.5 [A]. Answers vary by the exact value chosen, but on this specific, standardised, two-parameter problem the dynamics are governed exactly by the contraction factor $|1 - 2\alpha|$ (per Exercise 4.4's finding that this problem's Hessian is exactly $2I$). Any α between 0 and 1 converges, at different speeds. $\alpha = 0.3$ or $\alpha = 0.7$ (contraction 0.4) reach within 1% of optimal by epoch 5; $\alpha = 0.2$ or $\alpha = 0.8$ (contraction 0.6) by epoch 8; $\alpha = 0.05$ (contraction 0.9, close to the chapter's own $\alpha = 0.1$) takes about 36 epochs. At exactly $\alpha = 1.0$, the contraction factor is $|1 - 2| = 1$: the loss neither grows nor shrinks, oscillating at a fixed distance from the optimum forever rather than converging. Only once α exceeds 1.0 does the contraction factor exceed 1 and the loss diverge, exactly as $\alpha = 1.1$ does in Exercise 4.3. So the real dividing line between "just right" and "too large" on this problem sits at $\alpha = 1.0$, not a fuzzy "1.0-1.2" range. The qualitative marker of "too large" is loss *increasing* rather than decreasing over the run, not merely converging slowly.

Exercise 4.6 [E]. A reasonable experiment: train the same model twice, once with plain SGD and once with Adam. Sweep the initial learning rate over several orders of magnitude (e.g. 10^{-5} to 10^{0}) for each. Plot final loss (or validation accuracy) against the chosen learning rate for both optimisers. The claim predicts Adam's curve should stay flat and near-optimal across a wide band of learning rates, while SGD's curve should be sharply peaked: good only in a narrow band, diverging above it and painfully slow below it. The claim would be falsified if Adam's performance instead degraded about as sharply as SGD's outside some narrow band, i.e. if the two curves had comparably narrow "good" regions.

Exercise 4.7 [E]. A reasonable response: memory isn't the problem. L-BFGS-B keeps only the last few gradient differences, about 10 pairs, so at 50 million parameters it needs roughly 20 vectors × 50M × 8 bytes, about 8 GB. That fits on one GPU. The problem is the gradients. Its 2-iteration speed came from exact, full-batch gradients on a smooth convex problem. A neural network is trained on mini-batches, because the training data rarely fits in memory for a full pass, and mini-batch gradients are noisy. That noise breaks L-BFGS-B's line search and its curvature estimates. For SGD and Adam the same noise helps, because it pushes the optimiser past saddle points on a non-convex loss surface. So

the fast convergence you saw doesn't transfer, and mini-batch SGD or Adam is the right tool here. The square-size Hessian argument applies to exact Newton's method, which would need about 4,000 GB (3,725 GiB) at only 1 million parameters. It doesn't apply to L-BFGS.

Exercise 4.8 [R]. Discussion only. No single answer. Goh's piece frames momentum's benefit less as "escaping local minima via averaged-out noise" (the SGD story), and more as correcting for *ill-conditioning*. When a loss surface curves much more steeply in one direction than another (a narrow, elongated valley), plain gradient descent zig-zags across the narrow axis, while making frustratingly slow progress along the long axis. A single learning rate has to be small enough not to diverge on the steep direction, which caps progress on the shallow one. Momentum's accumulated velocity partially cancels the oscillation across the narrow axis, since opposing updates average out. It reinforces the consistent direction along the long axis, since aligned updates add up. That's a curvature-shape argument, distinct from and complementary to the "noise helps escape bad minima" story usually told about SGD. Worth noting the two mechanisms address different pathologies (conditioning vs. non-convexity).

Chapter 5 — Information Theory Basics

Exercise 5.1 [W]. Highest to lowest entropy: fair die ($H = 2.0000$ bits) > biased die $[0.70, 0.10, 0.10, 0.10]$ ($H = 1.3568$ bits) > constant die ($H = 0.0000$ bits). Entropy tracks how spread-out a distribution is. The fair die is maximally unpredictable: all four outcomes are equally likely. The biased die is more predictable, because one outcome dominates. The constant die has zero uncertainty, because the outcome is already known.

Exercise 5.2 [W]. $\mathrm{KL}(P\|Q)$ measures the expected extra cost of using Q when P is the true distribution, weighted by how probable each event is under P. Swapping P and Q changes which distribution's probabilities do the weighting, so the sums aren't the same. The rule for keeping it straight: the first argument is always "truth" (the actual/reference distribution), the second is always the "approximation" being scored against it.

Exercise 5.3 [A]. Both models classify every example correctly at the 0.5 threshold, but their losses differ because cross-entropy scores *confidence calibration*, not just which side of the threshold a prediction lands on. `y_pred_a = [0.6, 0.6, 0.4, 0.4]` is mildly confident about each correct call. Every term is $-\log(0.6) \approx 0.5108$ nats, so the average loss is exactly 0.5108. `y_pred_b = [0.99, 0.51, 0.49, 0.01]` is a mix: two predictions are barely on the correct side ($-\log(0.51) \approx 0.6733$ nats each, close to the $-\log(0.5) \approx 0.6931$ coin-flip baseline), and two are extremely confident and correct ($-\log(0.99) \approx 0.0101$ nats each). Averaging all four gives $(0.6733 + 0.6733 + 0.0101 + 0.0101)/4 \approx 0.3417$ nats: noticeably *lower* than model A's 0.5108, not similar or higher. The two barely-confident predictions cost close to the coin-flip baseline each, but the two extremely confident predictions cost almost nothing. Cross-entropy's cost for a correct call approaches zero as confidence approaches 1, with no floor above zero. That asymmetry means a model with some very confident correct calls can post a lower average loss than a model that plays it uniformly safe, even

while carrying a couple of near-toss-up calls. The lesson: 100% classification accuracy at a threshold says nothing about how good the underlying probabilities are. Cross-entropy does, and it doesn't always penalise a mixed-confidence model more than a uniformly moderate one.

Exercise 5.4 [A]. The entropy works out as

$$\begin{aligned} H([0.4, 0.4, 0.1, 0.05, 0.05]) &= -\big(2 \times 0.4 \log_2 0.4 + 0.1 \log_2 0.1 \\ &\qquad + 2 \times 0.05 \log_2 0.05\big) \\ &\approx 1.8219 \text{ bits.} \end{aligned}$$

The five-outcome maximum is $\log_2(5) \approx 2.3219$ bits, so this distribution sits exactly 0.5000 bits below maximum entropy. It's noticeably more uncertain than the biased four-sided die in the chapter, since probability mass is still split fairly evenly across the top two outcomes.

Exercise 5.5 [A]. For $P = [0.6, 0.4]$, $Q = [0.5, 0.5]$: $\mathrm{KL}(P\|Q) = 0.6 \log_2(0.6/0.5) + 0.4 \log_2(0.4/0.5) \approx 0.6 \times 0.2630 + 0.4 \times (-0.3219) \approx 0.1578 - 0.1288 = 0.0290$ bits. $\mathrm{KL}(Q\|P) = 0.5 \log_2(0.5/0.6) + 0.5 \log_2(0.5/0.4) \approx 0.5 \times (-0.2630) + 0.5 \times 0.3219 \approx -0.13152 + 0.16096 = 0.02945 \approx 0.0294$ bits. That's close to but not equal to $\mathrm{KL}(P\|Q)$, confirming the asymmetry again. The two happen to be numerically close here only because P and Q are close to each other. $H(P) = -\big(0.6 \log_2 0.6 + 0.4 \log_2 0.4\big) \approx 0.9710$ bits. $H(P) + \mathrm{KL}(P\|Q) \approx 0.9710 + 0.0290 = 1.0000$ bits. That matches $H(P, Q) = -(0.6 \log_2 0.5 + 0.4 \log_2 0.5) = 1.0000$ bits exactly. Since Q is uniform over two outcomes, $H(P, Q)$ reduces to exactly 1 bit regardless of P. The identity holds.

Exercise 5.6 [E]. A correlation of 0.1 and a mutual-information score of 0.1 are not on the same scale, and don't mean the same thing. Correlation is bounded in $[-1, 1]$, with well-understood conventions for "weak." Mutual information, in nats, as `mutual_info_classif` returns it, has no fixed scale. Its ceiling is the label's entropy, and its magnitude depends on the entropy of the specific label and feature distributions in that dataset. A score of 0.1 might be near the noise floor for one problem and a strong, useful signal for another, as the Worked Example showed with `feature_noise` scoring 0.043 against `feature_nonlinear`'s 0.338 on the same dataset. The chapter's Trade-offs section is explicit that MI should only be used to rank features *within* a single run, not compared against a fixed external threshold. The right move is to always rank features against each other (or against a known-noise baseline feature computed on the same data) rather than picking an absolute cutoff.

Exercise 5.7 [E]. Model answer: a binary label determines which of two *shapes* a numeric feature is drawn from, both centred at zero. For example, when label is 0, the feature is drawn from a standard normal. When label is 1, the feature is drawn from a mixture that is symmetric around zero but bimodal (e.g., a 50/50 mix of $\mathcal{N}(-2, 0.3)$ and $\mathcal{N}(2, 0.3)$). The mean of the feature is approximately zero regardless of the label, so correlation with the label is near zero. But the *shape* of the distribution, unimodal versus bimodal, differs sharply between labels. That's exactly the kind of higher-order dependence mutual information,

but not correlation, is sensitive to. This is a generalisation of the Worked Example's variance-based trick to an even more clearly non-linear signal (shape rather than spread).

Exercise 5.8 [R]. Discussion only. No single answer. Shannon's paper introduces channel capacity and the noisy-channel coding theorem, which this chapter never touches. The idea: a communication channel has a maximum rate at which information can be transmitted with arbitrarily low error, and entropy sets the fundamental limit on lossless compression. Worth considering how this connects to the chapter's ML framing: a classifier's cross-entropy loss floor at $H(P)$ is structurally the same idea as a channel's capacity limit. No amount of cleverness can push performance past a limit set by the intrinsic uncertainty of the source. Students should also notice that Shannon's original motivation was communication engineering, not statistics or machine learning, which came decades later and inherited the same mathematics.

Chapter 6 — Python Data Science Ecosystem

Exercise 6.1 [W]. pandas DataFrames store each column as a NumPy array under the hood, so pandas' speed and much of its behaviour (vectorised operations, dtype handling) come directly from NumPy. But NumPy arrays have no concept of labelled rows, mixed column types, or an index, so NumPy has no need to understand pandas at all. The dependency runs one way, matching the layered-stack diagram in the Exposition.

Exercise 6.2 [W]. `.fit()` learns parameters from training data. `.transform()` applies a learned transformation to produce new data, used for preprocessing. `.predict()` applies a learned supervised model to produce predictions on new data. `.transform()` and `.predict()` are safe to call on test data. `.fit()` should never be called on test data, since fitting on it leaks test-set information into the transformation or model.

Exercise 6.3 [A]. Timing `np.sqrt` against an equivalent `[math.sqrt(x) for x in py_list]` loop over 500,000 elements gives roughly 22.5 ms for the Python loop and roughly 0.46 ms for the NumPy call: a speedup of approximately **49x**. This is smaller than Listing 6.6's ~117x multiplication speedup, because `math.sqrt` is a fast built-in per call, narrowing (but not closing) the gap. The qualitative result, NumPy wins by well over an order of magnitude, holds regardless of which elementwise operation is timed.

Exercise 6.4 [A]. Pivoting survival rate by `pclass` and `embark_town` on the Titanic dataset shows the lowest survival rate is **third class passengers who boarded at Southampton**, at roughly **19%**, noticeably lower than third-class Cherbourg (38%) or Queenstown (38%) passengers. This is a case where a single grouping variable (class alone, or port alone) would have hidden the compounding effect visible only in the two-dimensional pivot.

Exercise 6.5 [A]. Comparing age between survivors (mean $\approx$ 28.3 years) and non-survivors (mean $\approx$ 30.6 years) with `scipy.stats.ttest_ind` gives `t ~ -2.07`, `p ~ 0.039`. That's below the 0.05 threshold, so the age difference is statistically significant, with survivors

being slightly *younger* on average. The effect is much weaker than the fare difference in Listing 6.8 (p ≈ 6e-15). Age separates survivors from non-survivors only modestly, while fare (as a proxy for class) separates them dramatically.

Exercise 6.6 [E]. If a scaler were fit on test data, or on the combined train+test set, rather than only on training data, the reported test accuracy would be inflated. The scaler's mean and standard deviation would already "know" something about the distribution of the test set. That gives the model an information advantage it wouldn't have in production, where future data is by definition unseen at fit time. No exception is raised because nothing about calling `.fit()` on the wrong slice of data is syntactically or semantically invalid. scikit-learn has no way to know which rows are supposed to represent "the future." So the leak is silent, and only shows up later as a mismatch between validation performance and real-world performance.

Exercise 6.7 [E]. Twenty columns of mixed numeric and categorical data is close to a worst case for raw NumPy. NumPy arrays are single-typed, so mixed columns would need to be split across multiple arrays, or coerced into a single dtype, usually `object`, which forfeits NumPy's speed advantage entirely. Every join, filter, or groupby that pandas expresses in one readable line would need to be hand-written with manual index bookkeeping. The "for speed" framing is backwards here: pandas' groupby and filtering operations already delegate to vectorised NumPy underneath. The actual speed cost of raw NumPy on this kind of data would be development time, not runtime.

Exercise 6.8 [R]. Discussion only, no single answer. Candidates worth investigating in the NumPy quickstart: the rule that dimensions are compared from the trailing (rightmost) axis, and either must match or one of them must be 1 or absent. Or that broadcasting can silently succeed on shapes the author didn't intend to be compatible: e.g. a `(3,)` array broadcasting against a `(3, 1)` array to produce a `(3, 3)` result via outer-product-like expansion, rather than the elementwise operation the author expected. Consider a concrete case: adding a "per-row mean" array of shape `(n,)` back onto a `(n, m)` matrix without reshaping it to `(n, 1)` first. Broadcasting rules make this fail loudly in some shapes and silently produce the wrong answer in others, depending on whether `n` happens to equal `m`.

Chapter 7 — Data Cleaning & Wrangling

Exercise 7.1 [W]. (a) MCAR: a hardware fault dropping readings at random has no relationship to the reading's value or any other variable. (b) MNAR: the probability of a missing income value depends directly on the (unobserved) income itself, higher earners are more likely to decline. (c) MAR: missingness correlates with an *observed* variable (signup channel), not with the true age value itself; conditional on signup channel, the missing ages aren't systematically different from the reported ones.

Exercise 7.2 [W]. The typo assignment `ages[0] = 312` is applied after the missing-value assignment, so that it deterministically overwrites row 0, regardless of whether index 0 happened to be selected by `np.random.choice(n, 40, replace=False)`. If the two lines were swapped, row 0 would first be set to 312. Then, only if index 0 were among the 40 randomly chosen missing indices, it would get overwritten back to `NaN`. That would make the presence of the planted outlier depend on random chance, rather than being guaranteed. The chapter's ordering guarantees the outlier exists every time the notebook runs with a fixed seed.

Exercise 7.3 [A]. The one-line change is `df['age'] = df['age'].fillna(df['age'].mean())` in place of `.median()`. After deduplication and before capping, age has a mean of 42.82 against a median of 42, the 312 outlier pulling the mean up above the rank-based median. Capping that single value down to 90 drags the mean back down to about 42.34, very close to the median of 42. So mean imputation would land at roughly 42.3, a third of a point above the reported median-imputed value of 42. The small remaining gap comes from the mild right skew still left in age even after the domain cap.

Exercise 7.4 [A]. `Q1` and `Q3` are the 25th and 75th percentile of the *whole* column. Only one value, 312, sits far above the bulk of the data (ages spread across 22–64). That single point has almost no leverage on the rank-based `Q1`/`Q3` statistics. 312 is one point out of hundreds, so the 25th/75th percentile boundaries barely move to accommodate it. The resulting IQR fence lands well below 312, meaning IQR *does* in fact catch it. That's the reported "1 outlier." The subtler risk is the opposite one: IQR is a whole-column, single-pass method. A handful of outliers or a long natural tail can each be caught or missed, depending on how many other values sit between the bulk of the distribution and the fence. That's why the chapter still applies a domain-knowledge cap (`[16, 90]`) on top of IQR, rather than trusting IQR's fence alone.

Exercise 7.5 [E]. A pipeline running without errors only confirms that every operation was syntactically and type-valid. It says nothing about whether the *result* is semantically correct. A concrete silent failure: an incorrectly specified `gender_map` that's missing the `'MALE'` (all-caps) variant wouldn't raise any error. `.map()` would convert every `'MALE'` entry to `NaN`, silently turning a known category into missing data. The pipeline would report success right up until someone later asks why 50 rows have a missing gender that used to have a value.

Exercise 7.6 [E]. Sample rules: (1) `(d['order_total'] > 0).all()`, no order should have a non-positive total; (2) `(d['discount_pct'] >= 0).all() and (d['discount_pct'] <= 100).all()`, a discount percentage must be a valid percentage; (3) `d['customer_id'].notnull().all()`, every order must be attributable to a customer; (4) a cross-column rule: `(d['delivery_date'] >= d['ship_date']).all()`, a package cannot be delivered before it ships. Single-column rules can never catch that, since both dates can individually look perfectly valid. A fifth candidate, if the data also had an `order_date` column: `(d['ship_date'] - d['order_date']).dt.days.between(0, 30).all()` as a sanity bound on fulfilment time, catching data-entry errors that swap order and ship dates.

Exercise 7.7 [R]. Discussion only, no single answer. Multiple imputation with a missingness indicator is the most tractable technique to start with. Alongside the imputed value, an explicit binary column recording *whether* the value was originally missing is added as a feature. That lets a downstream model learn that "this value was estimated" is itself informative, relevant when missingness correlates with the outcome, as in MNAR. A Heckman correction goes further, formally modelling a "selection equation" for whether a value is observed at all, jointly with the equation of interest. It requires an instrument variable that affects observability but not the outcome directly. That's often the hardest part to find in practice. Worth considering: unlike MCAR/MAR imputation, none of these techniques make the MNAR problem disappear. They make its assumptions explicit and testable rather than silently violated.

Chapter 8 — SQL for Data Science

Exercise 8.1 [W]. Standard SQL evaluates `HAVING` before column aliases defined in the `SELECT` list are resolved. The aggregate expression `SUM(revenue)` exists as soon as the grouping happens, but the alias `total_revenue` is only attached to the output column afterwards, so `HAVING` can't yet "see" it. PostgreSQL, SQL Server and Oracle follow the standard and reject the alias. DuckDB, MySQL and SQLite accept it as an extension. Repeating the full aggregate expression (`HAVING SUM(revenue) > 1000`) works because it doesn't depend on the alias having been assigned.

Exercise 8.2 [W]. With three rows tied for first place: `ROW_NUMBER()` breaks the tie arbitrarily (by whatever order the engine processes ties in) and assigns `1, 2, 3`. Every row gets a distinct number even though they're tied on the ordering column. `RANK()` assigns all three rows `1, 1, 1` and then the *next* row after them jumps to `4`, a gap appears that skips one fewer than the number of tied rows. `DENSE_RANK()` also assigns all three rows `1, 1, 1`, but the next row gets `2`, no gap. The choice matters operationally. `ROW_NUMBER()` is right for "give me exactly N rows": e.g. top 3, no ties allowed. `RANK()` is right when ties should be visible as ties, with the gap reflecting how many rows share that rank. `DENSE_RANK()` is right for "how many distinct rank levels are there": e.g. medal tables, ignoring how many people share a level.

Exercise 8.3 [A]. Adapting Listing 8.12 to `GROUP BY c.region` (joining `orders` to `customers` instead of `products`) gives, on the seed-42 dataset: East leads with $564,538.19 across 715 orders, West is a very close second at $564,314.41 across 666 orders, a difference of only $223.78. That's well within noise for a synthetic dataset of this size, and effectively a tie. North ($502,071.81, 557 orders) and South ($486,219.92, 591 orders) trail more noticeably. Unlike the category breakdown, where Sports led Clothing by a clear $76,509.76 margin, the regional split shows no real leader. East and West are statistically indistinguishable here, a reminder that "which group is #1" can be a fragile question when the top two are this close.

Exercise 8.4 [A]. Changing the retention window from 90 to 60 days in Listing 8.14 (same Q1 2022 cohort of 156 customers) drops the retained count from 128 (82.1%, the chapter's reported 90-day figure) to 99 (63.5%). That's an 18.6 percentage point difference. That sensitivity to the window length is evidence for the chapter's caution about the original 90-day figure. A retention rate that swings by nearly 19 points when the window shrinks by a third is a symptom of orders scattered close to uniformly across the observation period. As the Worked Example explains, orders were assigned to customers uniformly at random over 730 days. It isn't reflecting genuine, sustained re-engagement behaviour. A retention metric this sensitive to an arbitrary window choice needs the underlying data-generating assumptions checked before it's trusted at face value.

Exercise 8.5 [A]. Re-running Listing 8.13 over the full (unlimited) customer ranking shows the cumulative percentage crosses 50% at the 74th customer (49.68% at rank 73, 50.21% at rank 74). So 74 of the 200 customers, or 37.0% of the customer base, are needed to reach half of total completed revenue. That's close to, but slightly past, the "half the revenue from roughly a third of customers" pattern the chapter's Worked Example already flags as flatter than a genuine 80/20 Pareto split would produce.

Exercise 8.6 [E]. One concrete change: instead of drawing `quantity` uniformly from `randint(1, 6)` and `price` uniformly from `uniform(5, 500)`, draw per-customer purchase frequency from a **power-law / Zipf distribution**, e.g. `np.random.zipf(a=2, size=n_customers)`, to set how many orders each customer places. Combine that with a **log-normal price distribution**, `np.random.lognormal(mean, sigma)` instead of uniform prices, for products. Both changes matter for the same underlying reason. A Pareto (80/20) pattern emerges specifically from *multiplicative*, heavy-tailed processes: a small number of customers who both order far more frequently *and* skew towards higher-priced products. Independent uniform draws for quantity, price, and order assignment guarantee every customer's expected revenue converges towards the same value as sample size grows. That mechanically flattens the concentration curve, regardless of how many orders are simulated.

Exercise 8.7 [E].

```
SELECT
    category,
    SUM(CASE WHEN status = 'completed'
             THEN revenue ELSE 0 END) AS completed_revenue,
    SUM(CASE WHEN status = 'returned'
             THEN revenue ELSE 0 END) AS returned_revenue
FROM orders o
JOIN products p ON o.product_id = p.product_id
GROUP BY category;
```

This is preferable to two separate queries joined afterwards. It scans the `orders`/`products` join exactly once instead of twice. The database only needs to plan and execute a single query. The result arrives as one row per category, with both figures already aligned. No

separate join step is needed in pandas afterwards to line the two result sets up by category. It also guarantees consistency: both numbers are computed from the exact same snapshot of the underlying tables. Two independently-run queries executed moments apart are not strictly guaranteed to be, if the underlying data is being concurrently modified.

Exercise 8.8 [R]. Discussion only, no single answer. The logical evaluation order (`FROM` → `WHERE` → `GROUP BY` → `HAVING` → window functions → `SELECT` → `ORDER BY`) places window functions *after* `WHERE` and `HAVING`. By then, the row set is down to whatever rows and groups survive filtering. Window functions still run *before* the final `SELECT` list decides which computed columns to display. That ordering is why a window function can't appear inside a `WHERE` clause directly. At the point `WHERE` is evaluated, no window function has been computed yet. There's nothing for the filter to reference. It's also why filtering on a window function's result requires wrapping the whole query in a CTE or subquery: compute the window function inside, then filter with an outer `WHERE`. That's exactly the shape Listing 8.8 and procedures like Procedure 8.5 and the "top-N per group" skeleton in this chapter's Quick Reference use. Worth connecting this to why `HAVING` (Exercise 8.1) has the same "can't reference something not yet computed" character: both restrictions come from the same clause-ordering rule, just applied to aggregates versus window functions respectively.

Chapter 9 — Exploratory Data Analysis & Visualisation

Exercise 9.1 [W]. The five questions, in order: (1) what do I have (shape, types, nulls, duplicates); (2) what does each variable look like alone (distributions, outliers, skew); (3) how do variables relate to each other (correlations, conditional distributions); (4) how do variables relate to the target; (5) what surprises me. "What surprises me?" is placed last because it can only be answered after the first four questions have established a baseline of expectations. You can't recognise a surprise until you know what "normal" looks like for this specific dataset.

Exercise 9.2 [W]. A mean of 29.7 against a median of 28.0 means the mean is pulled upward relative to the median. That indicates a right-skewed (positive skew) distribution: a small number of unusually old passengers/customers/etc. stretching the upper tail. A histogram with a KDE overlay (Listing 9.4) would confirm the shape directly and show whether the skew is mild or extreme.

Exercise 9.3 [A]. Answer depends on the dataset chosen. The deliverable is a `quick_profile()` run and a reported list of columns exceeding 10% missing, following the exact output format of Listing 9.2. For example, on the Titanic dataset used in this chapter, only `deck` (77.2%) and `age` (19.9%) exceed that threshold.

Exercise 9.4 [A]. `df.groupby('embarked')['survived'].mean()` on the Titanic dataset returns roughly 55% survival for passengers who embarked at Cherbourg (C), 39% at Queenstown (Q), and 34% at Southampton (S). The spread (~20 points between C and S)

is large enough to be worth investigating as a feature. It is confounded with class, though: Cherbourg passengers were disproportionately 1st class, so some of this apparent effect is the class/fare effect from Listing 9.8 showing through a correlated variable.

Exercise 9.5 [A]. Running Listing 9.5's near-redundant-pair check on `survived, pclass, age, sibsp, parch, fare` finds no pair above the 0.95 threshold. The strongest pairwise correlation among the features (excluding the target) is `pclass`/`fare` at roughly −0.55, well below the redundancy cutoff. This dataset has essentially no redundant numeric feature pairs, so no columns need to be dropped purely on multicollinearity grounds.

Exercise 9.6 [E]. A plausible candidate is `fare` split by `pclass` and `survived` jointly. The marginal correlation between `fare` and `survived` (0.26) is real, but could hide a stronger within-class effect, since fare varies enormously *within* 1st class, from a low-end 1st-class ticket to the most expensive suites. A passenger paying a top fare within 1st class might show a bigger survival boost than the marginal correlation suggests. The general pattern to watch for: any feature whose marginal relationship to the target looks weak but which is known to vary systematically with a *third*, stronger predictor (here, class) is a candidate for the same hidden effect.

Exercise 9.7 [E]. Running EDA on the full dataset, test rows included, contaminates every pattern it finds. An outlier threshold, a "this feature looks predictive" judgement, or a bucketing scheme for rare categories has then been chosen with knowledge of the test set's actual values. Even without directly touching the labels, decisions like "drop rows where fare > 400" or "bucket embarked's rare category into 'other' " become choices tuned to the specific data points that will later be used to evaluate the model. That inflates the model's apparent generalisation performance. The fix is to split first, then run full EDA only on the training partition, treating the test set as if it doesn't exist until final evaluation.

Exercise 9.8 [R]. Discussion only, no single answer. Each of Anscombe's four datasets shares an identical mean, variance, and Pearson correlation coefficient (~0.816). Summary statistics alone would report the same numbers for all four, and suggest the same linear relationship strength. What they hide: one dataset is a genuine linear relationship with noise. One is a clean non-linear (quadratic-like) curve that Pearson's linear correlation badly misrepresents. One is a near-perfect linear relationship broken by a single outlier that shifts the fitted line. And one is a vertical cluster of points entirely determined by one high-leverage outlier. Only plotting each one reveals which of these four very different stories the identical statistics were describing.

Chapter 10 — The Data Science Lifecycle

Exercise 10.1 [W]. 1. Problem framing: Chapter 2. 2. Data collection & understanding: Chapters 7-9. 3. Data preparation: Chapters 7 and 14. 4. Modelling: Chapters 11-20. 5.

Evaluation: Chapters 13 and 16. 6. Deployment: Book 2, Chapter 23. 7. Monitoring: Book 2, Chapter 25. 8. Iteration: loops back to Chapter 2's problem framing.

Exercise 10.2 [W]. The left-to-right pipeline diagram implies each stage is finished once and handed off to the next, but real projects loop backward constantly: evaluation (stage 5) routinely reveals the success metric itself was wrong. A model hits a strong AUC, the team celebrates, and the business owner then explains they cared about precision at a specific operating point nobody optimised for, sending the project back to stage 1. Similarly, modelling (stage 4) routinely sends a project back to data preparation (stage 3) when a model's errors reveal a feature bug or missing signal. The two stages cycle many times rather than completing in sequence. A diagram with a single feedback arrow tacked on at the end can't represent a process where nearly every stage has an edge back to nearly every earlier one.

Exercise 10.3 [A]. Take this `ask` dictionary:

```
ask = {
    "simple_rule_suffices": False,
    "one_off_analysis_only": False,
    "historical_labelled_data_exists": False,
}
```

It describes, say, a stakeholder who wants to predict equipment failures but has never logged which equipment failed. It returns `"Not ready -- this is a data collection project first, an ML project second"`. This is the correct outcome because Listing 10.3 checks `simple_rule_suffices` first (false here, so it doesn't short-circuit to "ship a rule"), then whether this is a one-off analysis (it isn't), then whether historical labelled data exists. It doesn't, so no amount of modelling skill can produce a supervised model yet. The correct next step is building a labelling/data-collection process, not starting a lifecycle project that has no label to train against.

Exercise 10.4 [A]. On a dataset with a 3% positive rate, the naive "predict majority class" baseline (always predict the negative class) scores **97% accuracy** by construction. It is the negative class's share of the data. That number is a poor argument for production use because the baseline achieves it while catching zero positives. Precision and recall on the positive class are both 0.00. The model is useless for any business action that depends on identifying positives. This is the exact scenario the tutorial's dummy classifier shows at 93% accuracy and 7% churn. A high accuracy figure on an imbalanced dataset says nothing about whether the model does the one thing it's being built for.

Exercise 10.5 [A]. The two stages to loop back to are stage 3 (data preparation) and stage 1 (problem framing/success metric). For stage 3: add richer features. The tutorial notebook works from only six raw features: support tickets, payment failures, login recency, tenure, plan type, product count. Engineering additional signals, such as usage trend over time, interaction-weighted engagement scores, or billing-cycle-relative features (Chapter 14), could

give the model more to separate churners from non-churners than a modest AUC of 0.638 suggests is currently available. For stage 1: renegotiate the success metric. Precision ≥ 0.40 at recall ≥ 0.60 may be unattainable with the signal currently available. The business owner may need to accept a different operating point, such as lower recall in exchange for usable precision. Or they may need a different intervention design, such as a broader, cheaper retention outreach that tolerates lower precision. The alternative is asking the model to hit a target the data can't support yet.

Exercise 10.6 [E]. Moving from nightly batch scoring to real-time scoring at the moment of a support contact changes all three elements the tutorial's Stage 6 plan assumed. The serving pattern shifts from a scheduled batch job over the full customer base to a request-response endpoint invoked synchronously (or near-synchronously) when a support interaction begins. The model needs to be behind a low-latency API, e.g. a SageMaker real-time endpoint, rather than a nightly Batch Transform job. Features need to be computable on demand rather than pre-joined the night before, which likely requires a feature store with fresh, queryable feature values, rather than a batch feature table. The latency budget tightens dramatically. Batch scoring can tolerate minutes to hours per run. A support-interaction-triggered prediction needs to return in well under a second, so it doesn't stall the support agent's workflow. This rules out any model or feature pipeline with expensive real-time joins or slow feature computation. The rollback plan also changes. A bad batch run can be discarded and rerun before anyone sees the output. A bad real-time model is affecting live support decisions immediately. So the rollback plan needs an instant traffic-shifting mechanism, e.g. an endpoint alias or canary rollback, rather than "rerun the job." It also needs a fallback response, such as defaulting to the old rule-based flag, if the real-time endpoint is unavailable or too slow.

Exercise 10.7 [E]. CRISP-DM under-specifies monitoring. Per Appendix A's Framework Comparison, it has no explicit monitoring phase at all, ending effectively at deployment. A team that "follows CRISP-DM" literally has no process step that requires anyone to watch the model after it ships. No dashboards, no alerts, and critically no named owner for production health. That explains exactly the pattern described: models get shipped and never revisited. TDSP would have required an explicit deployment-as-ongoing / customer-acceptance stage with monitoring dashboards, drift alerts, and, per Appendix A's RACI table, a named individual (not "the team") accountable for receiving those alerts and triggering retraining. The gap isn't a modelling failure. It's a process framework that never asked "who watches this after ship day."

Exercise 10.8 [R]. Discussion only, no single answer. Worth examining directly: the original CRISP-DM paper treats deployment as its sixth and final phase, with only a brief note that the process is "cyclic." It offers no dedicated deliverable, checklist, or owner for what an iteration requires. Iteration is a property asserted of the diagram, not a phase with its own exit criteria. TDSP's documentation, by contrast, explicitly draws the lifecycle looping back into Business Understanding, and ties that loop to concrete artefacts: updated project charters, retrained model versions. The difference suggests CRISP-DM, written

in 1996 for data-mining projects that often ended at a delivered analysis or report, didn't anticipate that "the model runs continuously in production and needs to be revisited" would become the normal case. TDSP, written in the cloud/MLOps era, treats the loop-back as the expected steady state of a live system, not an edge case.

Chapter 11 — Supervised Learning: Overview of Approaches

Exercise 11.1 [W]. (a) Spam detection using word-count features: Probabilistic (Naive Bayes). Its core assumption is that features are conditionally independent given the class, which is a reasonable approximation for bag-of-words counts given a topic. (b) Predicting house price from square footage and location: Linear. Its core assumption is that the target is well-approximated by a linear combination of the input features. (c) Classifying flowers from petal measurements using their five closest labelled neighbours: Distance-Based (KNN). Its core assumption is that nearby points in feature space share a label, so a new point's class is decided by a majority vote among its closest training examples.

Exercise 11.2 [W]. The equation is Expected Test Error = Bias^2 + Variance + Irreducible Error. A shallow tree can only express a small number of coarse if/else rules, so it's forced to represent the true decision boundary poorly no matter what training data it sees. That's high, stable bias. Letting the tree grow deeper adds more splits, so it can carve out increasingly fine-grained regions that match the true boundary more closely, which is exactly what lowers the bias term. But those extra splits are fit to the specific noise in the training sample, and a different training set would produce different deep splits, so the fitted function becomes unstable across resamples, which is exactly what raises the variance term. Flexibility is a dial on how much the model is allowed to bend to fit what it's shown. Bending further to reduce systematic error is the same mechanism that makes the fit more sensitive to which particular examples it saw.

Exercise 11.3 [A]. Four algorithms in the benchmark table show test accuracy exceeding CV mean accuracy: Logistic Regression (0.982 vs. 0.980, +0.002), SVM (0.982 vs. 0.971, +0.011), Random Forest (0.956 vs. 0.954, +0.002), and Decision Tree (0.912 vs. 0.910, +0.002). Three of those four gaps are tiny and identical, +0.002, negligible next to the CV standard deviations reported alongside them (±0.013 for Logistic Regression, ±0.023 for Random Forest, ±0.019 for Decision Tree). SVM's gap is larger, +0.011, but it's still comfortably inside its own CV standard deviation (±0.018), so it isn't evidence of anything unusual either. In every case the test score sits within one CV fold's worth of ordinary variation. This ordering tells you almost nothing about overfitting one way or the other. It just means this particular 80/20 test split happened to land on the easy side of average for these four models. A test score meaningfully *below* the CV mean, or a train/test gap computed directly (Procedure 11.4), is the diagnostic that signals overfitting. A small positive gap like this, even SVM's larger one, is noise, not evidence of a well-generalising model.

Exercise 11.4 [A]. A Decision Tree builds its predictions entirely out of axis-aligned splits on individual features. Its expressive power comes from chaining many such splits together: each additional feature gives it another axis to cut along, letting it carve out increasingly fine decision regions. Restricted to two features, the tree has nowhere left to turn. It can only ever partition a 2D plane with rectangular cuts, a drastic reduction in what shapes of boundary it can represent. Its accuracy collapses accordingly (0.912 → 0.789). Naive Bayes works completely differently: it multiplies together per-feature likelihoods under an independence assumption, so each feature contributes an essentially separate, additive piece of evidence to the final posterior. Dropping 28 of 30 features just means losing 28 pieces of independent evidence from a sum-like combination. The mechanism itself, multiply what evidence you have, still works exactly the same way with two features as with thirty. It just has less to work with. That's why Naive Bayes degrades gracefully (0.930 → 0.877) while the tree, which fundamentally needs many features to build its structure, degrades sharply.

Exercise 11.5 [E]. Rule out K-Nearest Neighbours first. The Trade-offs table already flags KNN as "slow at predict time," and the Worked Example gives the concrete number: 35.4 ms to serve predictions on the held-out test set of only 114 samples drawn from a 455-row training set. That's six times slower than the next-slowest algorithm and over sixty times slower than the fastest. KNN's prediction cost is $O(n)$ in the size of the training set (per the Exposition's KNN section, every prediction requires a distance computation against every stored training point). Scaling from 455 to 40,000 training rows makes that cost far worse, not better. The per-request latency would grow roughly in proportion to the extra data, blowing well past the 5 ms budget, even before accounting for the nonlinear-interaction requirement working against a distance metric already strained by many features. Random Forest or gradient-boosted trees, which handle nonlinear interactions natively and predict in single-digit milliseconds regardless of training set size, are the algorithms worth trying here.

Exercise 11.6 [E]. "No algorithm is better than any other" implies a working data scientist should be indifferent between models. That is false and dangerous advice in practice. On the concentric-circles dataset, Logistic Regression scored 0.289, worse than a coin flip, while Random Forest scored a perfect 1.000, on the *same* data. Two algorithms performed wildly differently on one concrete problem. Nothing about that result says "they're equally good on average across every problem." But it emphatically says they are not equally good *on this problem*, which is the one the data scientist has to solve. The narrower claim the theorem makes: averaged over *all possible problems*, including ones with no structure whatsoever, unlike anything encountered in practice, no algorithm has an edge. It says nothing about the much smaller, highly structured set of problems real datasets belong to, where algorithms whose inductive bias matches the data's actual structure, a linear model on linearly-separable blobs, a tree ensemble on nonlinearly-separable circles, reliably win.

Exercise 11.7 [R]. Discussion only, no single answer. Applying Burkov's framework to Naive Bayes: the **hypothesis representation** is a set of per-feature conditional probability distributions, one likelihood distribution per feature per class (for `GaussianNB`, a Gaussian

mean and variance for each numeric feature within each class). These are combined multiplicatively via Bayes' theorem to produce a posterior over classes. The **loss function** is essentially absent in the sense logistic regression uses one. Rather than defining an error to minimise and searching for parameters that reduce it, Naive Bayes' parameters are the maximum-likelihood estimates of each per-feature distribution, computed directly from counts or sample statistics. The **optimisation procedure** follows from that. The independence assumption reduces parameter estimation to closed-form per-feature statistics (a mean and variance per class, or a frequency count per category). So fitting is a single pass over the training data with no iteration. Contrast Logistic Regression, whose sigmoid-linked linear combination has no closed-form maximum-likelihood solution and requires an iterative convex optimiser (gradient descent or L-BFGS, controlled by `max_iter`) that repeatedly evaluates and adjusts every coefficient until convergence. That structural difference, closed-form counting versus iterative search, is exactly why Naive Bayes trains in under 2 ms while Logistic Regression takes over 15 ms on the same data, nearly a 10x gap. The gap widens sharply as feature counts grow.

Chapter 12 — Ensemble Methods

Exercise 12.1 [W]. If B models each have variance σ^2 and their errors are uncorrelated, averaging them gives variance σ^2/B, strictly smaller as B grows, with no term in that formula affecting bias. Bagging doesn't change what each individual model is trying to predict: no change to the systematic error, i.e. bias. It only reduces how much the ensemble's *average* prediction fluctuates around that target across different training samples.

Exercise 12.2 [W]. (a) Boosting: a model systematically wrong on a subgroup has high bias on that subgroup, which boosting directly targets by reweighting/refitting on exactly the hard cases. (b) Bagging: wildly changing predictions from small training-data changes is the definition of high variance, which bagging's averaging over bootstrap samples is designed to reduce.

Exercise 12.3 [A]. Random Forest's RMSE barely moving from 200 to 500 trees ($45,502 → $45,514, marginally *worse*, well within noise) indicates bagging's variance reduction has already saturated before 200 trees: additional trees add averaging stability with diminishing, essentially flat returns. LightGBM keeps improving slightly over the same range ($28,323 → $28,159), because boosting is still reducing bias round by round at 200 trees. Each new tree is still correcting real signal the ensemble hasn't captured yet, not just fitting noise. That's the core asymmetry this chapter's Exposition draws between the two. Bagging's averaging effect plateaus once the ensemble is large enough to cancel variance. Boosting keeps extracting value from additional rounds for longer, up to the point, not yet reached here, where it starts overfitting instead.

Exercise 12.4 [A]. LightGBM, not XGBoost, is the single best individual model in this chapter's benchmark: $28,323 RMSE, versus XGBoost's $28,873. It's already one of the

two models in the existing stack. That makes beating solo XGBoost's $28,873 specifically a fairly low bar: LightGBM alone already clears it, so the three-model stack would need the meta-learner to badly mis-weight things for the combined prediction to land worse than $28,873. The more interesting, harder bar is solo LightGBM's own $28,323. This chapter's own measured result is a reason for caution there: the RF+LightGBM stack's meta-learner assigned Random Forest, the weaker base model, a *negative* weight, and the stack still ended up worse than solo LightGBM ($29,535 vs. $28,323). Adding XGBoost, weaker than LightGBM but stronger than Random Forest, could get down-weighted the same way and land close to solo LightGBM, or dilute it enough to land worse. Expect the three-model stack to beat solo XGBoost comfortably. Don't assume it beats solo LightGBM without checking.

Exercise 12.5 [E]. "Which models were in the stack, and did you compare against the single best individual model before concluding the stack is better? This chapter's own measured result shows a stack can be worse than the best model it was built from, even with that model inside the stack. 'Stacked and improved over all five individually' needs to be checked against the best of the five, not just an average or the worst of them."

Exercise 12.6 [E]. Random Forest-first is right when time-to-baseline matters more than squeezing out the last accuracy, since it needs almost no tuning and gives a solid, honest number quickly (per Procedure 12.3). It's the wrong default when the roughly 58% RMSE gap this chapter measured (Random Forest $45,502 vs. XGBoost $28,873) matters to the business outcome. At that point, the extra tuning effort XGBoost/LightGBM demand is worth paying for, and "simpler" shouldn't be conflated with "good enough."

Exercise 12.7 [R]. Discussion only. Likely candidates include XGBoost's cache-aware block structure for out-of-core computation (enabling training on datasets larger than memory) or its weighted quantile sketch algorithm for approximate split-finding on weighted data. Either addresses a scalability problem (very large datasets, or weighted training samples) beyond what this chapter's Trade-offs table, which focuses on bias/variance/overfitting risk, covers.

Chapter 13 — Cross-Validation & Model Selection

Exercise 13.1 [W]. Choosing hyperparameters based on test-set performance means the model was implicitly optimised, at least in part, to do well on that exact set of examples. The test set has stopped being purely "unseen data." It has become part of the selection process. Any resulting test-set score is now measuring how well the model fits that specific set, not how well it generalises to new data.

Exercise 13.2 [W]. (a) Leave-One-Out: 30 patient records is well under the "<50 examples" threshold for LOO. (b) TimeSeriesSplit: daily sales are strictly time-ordered, so random

shuffling would leak the future into training. (c) Stratified k-fold (k=5): a roughly balanced, moderate-size dataset is the standard case Stratified k-fold with k=5 is designed for.

Exercise 13.3 [A]. The ten seed-accuracy values have a standard deviation of about 0.010 (`np.std`), roughly 1 percentage point. That's somewhat *larger* than the 5-fold CV standard deviation of 0.007. The 5-fold CV estimate is still the more trustworthy of the two, and not just because its std happens to be smaller. It's built from five disjoint folds that together cover the whole dataset exactly once, systematically. The ten single-split values, by contrast, are ten separate 80/20 splits that overlap heavily. They aren't independent measurements of the same underlying quantity, not in a structured resampling sense. A single seed's score depends entirely on which specific examples happened to land in that one test set. CV's reported score is already an average over five such sets. That's exactly why it ends up both more stable and more representative than any pile of ad hoc reshuffles, however many are tried.

Exercise 13.4 [A]. A gap of zero, or effectively zero, between non-nested and nested estimates (like this chapter's 0.9543 vs. 0.9543 result) indicates the optimism bias is negligible for this dataset/search combination. That bias comes from selecting hyperparameters on the same data used to evaluate them. Either the dataset is easy enough to separate, or the hyperparameter search space is small enough, that overfitting to the inner CV folds isn't a serious problem here. A gap of several percentage points would indicate the opposite. The non-nested number would be meaningfully inflated by selection bias. Only the nested estimate should be reported or trusted as the honest expected performance.

Exercise 13.5 [E]. CV std tells you how much the score varies across folds of *this* dataset with *this* model. It does not tell you whether the underlying data collection process itself is representative of the deployment population. It does not tell you whether there's target leakage baked into the features. It does not tell you whether the model will face distribution shift once deployed. A tight CV std is consistent with a reliable estimate on this data. But it says nothing about whether this data is the right data to be estimating from.

Exercise 13.6 [E]. "Regular `StratifiedKFold` on time-ordered data allows validation folds that sit chronologically *before* some of the training data within the same fold rotation. That means the model may have effectively seen future information relative to what it's being scored on. The strong reported result is very likely optimistic and would not hold up once the model only has access to past data at deployment time. I'd ask them to rerun with `TimeSeriesSplit` before trusting that number."

Exercise 13.7 [R]. Discussion only. `cross_validate` returns fit time and score time per fold (and can return train scores alongside validation scores, and multiple metrics simultaneously) that `cross_val_score` does not. This chapter's Procedure 13.5 (comparing two models) would benefit from `cross_validate`'s per-fold timing when training cost, not just accuracy, is part of deciding between two otherwise similarly-accurate models.

Chapter 14 — Feature Engineering

Exercise 14.1 [W]. Tree-based models make decisions by splitting on a threshold ("is `age` > 35?"), and the split location adapts to whatever scale the data is already in. Multiplying a feature by 1000 just shifts where the threshold lands, without changing which points fall on which side. Linear and distance-based models combine features directly instead: weighted sums, Euclidean distances. A feature on a much larger scale mechanically dominates the combination or distance calculation, regardless of its actual predictive value, unless scaled to a comparable range first.

Exercise 14.2 [W]. (a) Ordinal encoding: `small`/`medium`/`large` has a genuine order. (b) Target or frequency encoding (CV-safe): 50,000 unique values feeding a linear model rules out one-hot (too many columns) and label encoding (implies a false order the linear model would misread). (c) One-hot encoding: 15 unique values, no natural order, is squarely in one-hot's low-cardinality nominal range.

Exercise 14.3 [A]. At hour 23, $\text{hour_sin}(23) = \sin(2\pi \cdot 23/24) \approx -0.259$, $\text{hour_cos}(23) = \cos(2\pi \cdot 23/24) \approx 0.966$. $\text{hour_sin}(0) = 0$, $\text{hour_cos}(0) = 1$. Euclidean distance $\approx \sqrt{(-0.259 - 0)^2 + (0.966 - 1)^2} \approx \sqrt{0.067 + 0.001} \approx 0.26$, small, correctly reflecting that hour 23 and hour 0 are only one hour apart, versus the raw integer difference of 23.

Exercise 14.4 [A]. Replacing `customer_id` with a column that directly (if noisily) encodes the label would very likely make the leaky-vs-safe gap grow substantially larger than 0.872 vs. 0.870. Per the Worked Example's explanation, leakage's damage scales with how strongly the leaking column correlates with the target once leaked. A column that's essentially a disguised version of the label is about as strongly correlated as a feature can be. Naive (leaky) encoding would likely produce a dramatically inflated CV score that collapses on the true held-out test set, unlike the mild, honest gap measured for the weakly-predictive `customer_id`.

Exercise 14.5 [E]. "8,000 one-hot columns will blow up training time and memory for very little benefit. Most product IDs will appear too rarely for the model to learn anything reliable about that specific dummy column, and you'll be feeding the model 8,000 mostly-empty columns instead of a compact signal. Frequency encoding or CV-safe target encoding captures the useful signal in one column instead."

Exercise 14.6 [E]. A column like `last_order_value_rounded` (or any feature that's derived from, or closely tracks, the order value itself after the fact) would be a strong candidate. Say such a column existed and were naively target-encoded, or even just included as a raw feature, without checking whether it's available *before* the order value is known. It would very likely produce a large, disqualifying leaky-vs-safe gap, since it's effectively leaking the label back into the features.

Exercise 14.7 [R]. Discussion only. `sklearn`'s built-in `TargetEncoder` (added in recent sklearn versions) performs its cross-fitting internally and automatically during `.fit_`

`transform()`. It chooses fold splits and smoothing parameters without the user hand-writing the loop. This removes the risk of a manual implementation bug, such as an off-by-one in fold indexing, that could silently reintroduce leakage in a hand-rolled version like Procedure 14.2's. One trap remains. On training data, call `.fit_transform()`, which cross-fits. Calling `.fit()` then `.transform()` on the same training data skips the cross-fitting and leaks.

Chapter 15 — Feature Selection & Dimensionality Reduction

Exercise 15.1 [W]. Feature selection removes features but keeps the survivors as-is, so "income" in the output is still literally income. Its scale, units, and meaning are unchanged, and a domain expert can interpret it directly. Feature extraction (PCA) creates new features as linear combinations of many originals. A principal component has no inherent real-world meaning of its own, only a mix of loadings across original features.

Exercise 15.2 [W]. (a) Variance threshold: filter (no model involved, judges only the feature's own spread). (b) RFECV: wrapper (repeatedly retrains a model and cross-validates to pick a feature count). (c) Random Forest importance above the median: embedded (importance is a byproduct of fitting one model, not a separate model-agnostic step or repeated retraining).

Exercise 15.3 [A]. With `k=5`, `SelectKBest(mutual_info_classif, k=5)` keeps the 5 highest-MI-scoring features. Mutual information scores are computed once and ranked, so the top 5 are necessarily a subset of the top 10, which is itself a subset of any top-15 selection. The ranking doesn't change with `k`. Only how far down that ranking you cut does. This holds only because the scores are seeded. `mutual_info_classif` is randomised, which is why Listing 15.3 passes `random_state=42`.

Exercise 15.4 [A]. Running the same `cumsum(pca_full.explained_variance_ratio_)` logic with the threshold raised to 0.99 gives 39 components, only 2 more than the 37 needed for 95%. That's a smaller jump than the "climbing the last few per cent gets expensive" intuition might suggest. It's a direct consequence of this dataset's structure: 20 of the 50 features carry genuine signal (10 informative, 10 redundant combinations of them) and 30 are pure noise. Once PCA has captured the signal-carrying components, the remaining components are splitting up noise variance that's spread almost evenly across the leftover dimensions, so each one adds only a sliver past the point where the real structure is already represented. A dataset with a longer tail of informative-but-individually-small directions would show a bigger 95%-to-99% jump than this one does.

Exercise 15.5 [A]. Lowering `perplexity` from 30 to 5 tells t-SNE to weight a much smaller local neighbourhood per point. The resulting plot typically shows tighter, more numerous, more fragmented small clusters. Local structure within digit classes becomes more visible, but the plot loses some of the smooth global separation between whole digit classes that a larger perplexity provides.

Exercise 15.6 [E]. In high dimensions, many features are noisy or redundant. A model can fit spurious patterns in that noise, which is overfitting, and irrelevant dimensions make distance-based reasoning less meaningful, the curse of dimensionality. Removing such features doesn't just save compute. It removes the substrate the model could otherwise overfit to, which is why validation accuracy can rise, not just hold steady, after selection.

Exercise 15.7 [E]. PCA components are linear combinations of many original applicant-provided variables: income, age, credit history, and so on, all blended with different weights. A regulator asking "why was this applicant denied" cannot be given an answer in terms of "component 7," not in any way a human applicant could act on or an auditor could verify against the original variables. Feature selection, keeping a subset of the 200 original variables via RFECV or embedded importance, preserves that direct explainability and should be recommended instead.

Exercise 15.8 [R]. Discussion only. Wattenberg et al.'s piece emphasises that t-SNE cluster *sizes* and *inter-cluster distances* in the resulting plot are not reliably meaningful. Only relative local neighbourhoods are. Applied to the digits plot: a viewer should resist reading "digit class 3's cluster is bigger than class 7's" or "class 1 is closer to class 4 than to class 9" as quantitative facts, even though the plot invites exactly that reading.

Chapter 16 — Model Evaluation

Exercise 16.1 [W]. A false positive is a legitimate email wrongly flagged as spam: annoying, since a real message gets buried, but rarely dangerous. A false negative is actual spam that gets through to the inbox: usually mildly annoying and cheap, though a phishing email that gets through can be costly. Between the two, false positives are generally treated as more costly for a spam filter. Losing a legitimate email, a job offer, a bill, can have real consequences, while an occasional spam message reaching the inbox is a minor inconvenience.

Exercise 16.2 [W]. (a) Recall: missing a real disease case (false negative) is the costly error in screening. (b) Precision: flagging a legitimate email as spam (false positive) is the costly error. (c) F1: with no stated cost asymmetry, a balanced single metric is the reasonable default for general comparison.

Exercise 16.3 [A]. Precision $= TP/(TP + FP) = 71/(71 + 1) = 71/72 \approx 0.9861$. Recall $= TP/(TP + FN) = 71/(71 + 1) = 71/72 \approx 0.9861$. Both match the reported figures. In this case, precision and recall come out numerically equal because FP and FN happen to both equal 1 on this particular test set.

Exercise 16.4 [A]. A model predicting "fraud" for every transaction in the 0.1%-fraud scenario would be correct only on the 0.1% that are fraud, giving accuracy ≈ 0.001 (0.1%), catastrophically low. Precision would be $TP/(TP + FP) = 0.001$, since only 0.1% of its "fraud" calls are actually fraud. Recall would be 1.0: it catches every real fraud case,

trivially, by flagging everything. This is the mirror-image failure mode to the "always predict not-fraud" model: perfect recall bought at the cost of near-total precision collapse.

Exercise 16.5 [E]. "99.9% accuracy on a fraud dataset where fraud is rare is close to what you'd get by predicting 'not fraud' every single time. Accuracy alone can't distinguish a good model from a completely useless one under this kind of imbalance. Before agreeing it's ready, I'd want to see recall and precision (or PR-AUC) specifically on the fraud class, since those are the numbers that actually say whether it catches real fraud."

Exercise 16.6 [E]. Random Forests are a common source of S-shaped miscalibration, under-confident at the extremes. A `predict_proba` of 0.75 might correspond to an observed positive rate closer to 0.9. Using such probabilities directly to set premiums risks systematically mispricing risk at the extremes: charging too little for the genuinely high-risk cases, and too much for the lowest-risk ones. Before trusting the numbers, plot a reliability diagram (Procedure 16.4). If it deviates from the diagonal, apply `CalibratedClassifierCV` with isotonic regression before using the probabilities for pricing.

Exercise 16.7 [R]. Discussion only. Davis and Goadrich's core claim is that ROC curves can present an overly optimistic picture of classifier performance specifically under class imbalance. The false positive rate, the ROC curve's x-axis, is normalised by the large number of true negatives, making even a fairly high absolute count of false positives look small. Precision, the PR curve's y-axis, is by contrast directly sensitive to that same false positive count relative to total positive predictions. This is exactly why this chapter recommends PR-AUC over ROC-AUC for rare-event detection: PR-AUC doesn't get the same optimistic distortion from an abundance of true negatives.

Chapter 17 — Hyperparameter Optimisation

Exercise 17.1 [W]. $5^5 = 3{,}125$ because grid search enumerates every combination across all 5 hyperparameters' 5 values each, with no mechanism to notice or exploit the fact that only some of them matter. It treats every dimension as equally worth exhaustive coverage. Even if only 2 of the 5 affect performance, grid search still multiplies out all 5 dimensions blindly. It has no way to detect in advance which ones are the unimportant ones.

Exercise 17.2 [W]. (a) Grid search is reasonable: 2 hyperparameters and sub-second training keeps even exhaustive coverage cheap, and grid search's full, reproducible coverage has no real downside at this scale. (b) Bayesian optimisation (Optuna) is the better default: 8 hyperparameters and 20-minute training runs make each evaluation expensive, exactly the regime where sample efficiency (fewer wasted trials) matters most.

Exercise 17.3 [A]. The gap is $0.8942 - 0.8933 = 0.0009$ (0.09 percentage points) in random search's favour. It's small, but it isn't noise. It traces directly back to the grid's construction. With only two values per dimension across six hyperparameters, every one of grid search's 64 combinations sits at a corner of the search space: always the minimum or maximum

`subsample`, `max_depth`, and so on. It can never place a trial in the interior of any dimension. Random search samples continuously across the same six ranges, so it can land on interior values the grid structurally excludes. It does: its winning configuration used `max_depth=6` and `subsample~0.88`, values no combination in the 64-corner grid could ever produce.

Exercise 17.4 [A]. Sketch:

```
def objective(trial):
    params = {...}
    model = xgb.XGBClassifier(
        **params, random_state=42, verbosity=0
    )
    for round_num in range(0, params['n_estimators'], 10):
        # train incrementally / use eval_set with early rounds
        # validation AUC after this many rounds
        intermediate_auc = ...
        trial.report(intermediate_auc, step=round_num)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()
    return final_auc

study = optuna.create_study(
    direction='maximize',
    pruner=optuna.pruners.MedianPruner(
        n_startup_trials=5, n_warmup_steps=10
    ),
)
```

Trials tracking below the median of previous trials at the same step get abandoned early, saving the remaining boosting rounds' compute for more promising configurations.

Exercise 17.5 [E]. "Optuna's sample efficiency pays off when you use it as a sample-efficient method: stop once it finds a good-enough configuration, rather than forcing it through the same trial budget as a cruder strategy. In this chapter's matched comparison, Optuna needed only 5 trials (5.7 seconds) to reach the CV AUC that random search needed its full 64 trials (18.5 seconds) to find. It went on to find the best score of the three strategies by the time its own 64 trials were up. But that full run took 171.5 seconds, more than the other two combined, because nobody told it to stop once it had already won. 'Always use Optuna' is right about its accuracy and its trial efficiency. It's wrong if the trial budget gets copied from a strategy that doesn't need to be stopped early."

Exercise 17.6 [E]. "Was that 0.97 validation AUC measured on the same folds used during the hyperparameter search, or on a held-out test set that never participated in tuning? If it's the former, the number is optimistic by construction: the search process selected for whichever configuration happened to score best on those specific folds, and the honest number is whatever a truly untouched test set reports."

Exercise 17.7 [R]. Discussion only. Bergstra and Bengio's argument is strongest in higher-dimensional search spaces, where only a small effective subset of hyperparameters meaningfully affects the objective, and grid search is additionally restricted to just a couple of values per dimension. The more total dimensions, and the coarser grid search's coverage of each one, the more its exhaustive-but-uniform corner sampling is wasted relative to a method that can reach the interior. This chapter's own comparison used six hyperparameters with a grid restricted to two boundary values each, exactly that regime. It showed the predicted gap. Random search beat grid on both CV and test AUC, and Optuna beat both by using its trial history to guide where it sampled next. It also showed a side of the argument Bergstra and Bengio don't dwell on: Bayesian optimisation's real advantage over random search shows up in trials needed, not in a matched-budget wall clock.

Chapter 18 — Regularisation

Exercise 18.1 [W]. Ridge's penalty ($\sum w_i^2$) has a smooth, round constraint boundary with no corners. The optimal solution under gradient descent settles at whatever point balances fit against penalty smoothly: weights shrink continuously towards zero but rarely land exactly on it. Lasso's penalty ($\sum |w_i|$) has a diamond-shaped constraint region with corners sitting exactly on the coordinate axes. The loss-minimising solution frequently lands precisely on one of those corners, which corresponds to a coefficient of exactly zero.

Exercise 18.2 [W]. This is a variance problem: training error is low (1.2) but test error is much higher (3.1), the classic large train/test gap signature of overfitting (high variance), not underfitting (which would show high error on both). Any of Ridge, Lasso, Elastic Net, dropout (if a neural net), or early stopping would help.

Exercise 18.3 [A]. Unregularised ratio: $1.912/1.412 \approx 1.35$. Test error is about 35% higher than training error, a real but modest overfitting gap given how small this dataset is. Ridge ratio: $1.461/1.819 \approx 0.80$. Test error is *below* training error. A ratio below 1.0 shows the model isn't just avoiding overfitting. It's generalising better than its own training performance would suggest. That can happen when the test split happens to be marginally "easier," or when the regularised model is closer to the true underlying function than to the noisy training sample.

Exercise 18.4 [A]. The true data-generating process is $y = 0.5x^2 - x + 2$ plus noise: a quadratic, needing only an x^2 term and an x^1 term (plus the intercept, handled separately) to represent exactly. A degree-15 polynomial expansion has 15 non-intercept terms. Lasso zeroing 11 of them and keeping 4 is broadly consistent with the model identifying that only a couple of low-degree terms carry the real signal. The eliminated terms exist only to fit noise, which Lasso's sparsity mechanism is specifically built to remove. The 2 extra nonzero terms beyond the ideal count aren't a contradiction. With only 42 training points, cross-validation picked whichever alpha minimised validation error, not whichever alpha recovered the exact

ground-truth term count. A small amount of residual noise-fitting surviving alongside the real quadratic signal is expected.

Exercise 18.5 [E]. Setting `alpha=100` without cross-validating risks landing far past the sweet spot into the underfitting regime. An arbitrarily large, unvalidated α can suppress real signal along with noise. The result is a model that's "safe" from overfitting only because it's no longer capturing the pattern at all, with both training and test performance suffering together. The Trade-offs section's guidance is explicit: find the sweet spot via cross-validated search, not by picking a large number and hoping.

Exercise 18.6 [E]. The feature in the thousands will have its weight naturally driven small by ordinary least-squares fitting, to keep its large-magnitude contribution to predictions reasonable. The L2/L1 penalty, which acts on raw weight magnitude, barely touches it as a result. The 0-to-1-scaled feature's weight, meanwhile, will naturally be larger to have any effect at all, so it absorbs a disproportionate share of the regularisation penalty. The practical effect: the large-scale feature is under-regularised, and the small-scale feature is over-regularised, purely as an artefact of scale rather than actual relevance. That's exactly why scaling must always precede regularisation.

Exercise 18.7 [R]. Discussion only. In the Gaussian-prior interpretation, the prior's variance σ^2 and Ridge's α are inversely related. A smaller prior variance expresses stronger belief that weights should be close to zero, which mathematically corresponds to a larger penalty (larger α) pulling the MAP (maximum a posteriori) estimate towards zero more forcefully. A very small prior variance is a strong prior belief in small weights, exactly mirroring a large α's strong regularisation pressure.

Chapter 19 — Unsupervised Learning

Exercise 19.1 [W]. K-Means's algorithm requires K centroids to initialise and assign points to before it can even begin iterating. The number of centroids is a structural input to the procedure, not something it discovers. DBSCAN instead defines clusters implicitly, as whatever dense regions emerge from the `eps/min_samples` density criteria applied to the actual data. However many dense regions exist is however many clusters result, with no upfront count required.

Exercise 19.2 [W]. (a) K-Means: large, roughly spherical, well-separated segments are exactly K-Means's best-case assumption, and it scales better than DBSCAN on large data. (b) DBSCAN: an unknown cluster count with irregular shapes and genuine noise is exactly the case K-Means structurally cannot handle (fixed K, spherical assumption, no built-in outlier concept) and DBSCAN is built for.

Exercise 19.3 [A]. Silhouette score measures how well each point fits its *assigned* cluster relative to the next-nearest cluster, using whatever clustering was produced. It has no access to the true crescent-shaped structure, and cannot detect that K-Means drew the wrong kind

of boundary. A K-Means clustering on the moons data could report a moderate-to-decent silhouette score for the (wrong) roughly-circular split it found. That's precisely because silhouette only asks "is this point closer, on average, to the points in its own cluster than to the points in the nearest other one," which K-Means's convex clusters can satisfy reasonably well even while completely missing the true crescent shapes. Silhouette validates internal consistency with the algorithm's own assumptions, not correctness against ground truth.

Exercise 19.4 [A]. With 27 flagged, precision 0.852 means $27 \times 0.852 \approx 23$ true positives, so false positives $= 27 - 23 = 4$. Recall 0.920 on 25 true anomalies means $25 \times 0.920 = 23$ true positives (consistent), so false negatives $= 25 - 23 = 2$. This run made more false positives (4) than false negatives (2): it over-flagged slightly more than it missed, though both error types are present.

Exercise 19.5 [E]. "How well do those 6 clusters hold up when you show them to someone who knows these customers? Do the groups correspond to real, distinct behaviours, or just to a geometrically tidy split of the feature space? A good silhouette score tells you the clusters are internally compact and well-separated; it says nothing about whether they're the *useful* 6 groups for your business decisions."

Exercise 19.6 [E]. Setting `contamination` to whatever value produces the "cleanest-looking" separation is circular, because it's using the model's own output to justify the input that produced that output. Instead of `contamination` reflecting an actual prior belief about how much of the data is truly anomalous, it's being reverse-engineered from a desired visual result. This risks hiding real anomalies (if the "clean" contamination is set too low), or manufacturing fake ones (if set too high), rather than reflecting genuine fraud prevalence.

Exercise 19.7 [R]. Discussion only. The Isolation Forest paper's key complexity claim is that it achieves roughly linear time complexity and constant memory with respect to dataset size. That's because it doesn't need to compute pairwise distances or density estimates the way distance-based methods (like LOF or One-Class SVM) do: isolation only requires random splits, not distance comparisons between all pairs of points. This directly explains this chapter's "scales well, works in high dimensions" summary. Avoiding pairwise distance computation is exactly what lets it scale where density- or distance-based alternatives struggle.

Chapter 20 — Time Series Analysis

Exercise 20.1 [W]. Earlier chapters' datasets were assumed independent and identically distributed. Each row's order carries no information, so shuffling before splitting doesn't change what the model can learn. Time series data has the opposite property by definition: today's value depends on yesterday's. A random split lets some training rows sit chronologically after some test rows, letting the model implicitly "see" future information relative to what it's being scored on.

Exercise 20.2 [W]. A slowly-decaying ACF (no sharp cutoff) is the signature of non-stationarity: the series' statistical properties are drifting over time rather than settling into a stable pattern. The action it suggests is differencing the series (and re-running the ADF test) before attempting to fit ACF/PACF-based model orders or an ARIMA model at all.

Exercise 20.3 [A]. Rejecting the null hypothesis at $p \approx 3 \times 10^{-27} < 0.05$ means there's overwhelming statistical evidence against "this series has a unit root," i.e. is non-stationary. The differenced series can be treated as stationary, satisfying the assumption ARIMA-family models need. The raw series' $p = 0.682$, far above 0.05, means the null hypothesis of non-stationarity cannot be rejected. That's consistent with the visible upward trend that was deliberately built into the synthetic data.

Exercise 20.4 [A]. Naive MAE (4.602) is lower than ARIMA MAE (7.228) by $(7.228 - 4.602)/7.228 \approx 36.3\%$: naive outperforms ARIMA by roughly 36% on this metric. A longer horizon (say, 180 or 365 days) would more plausibly reverse this result. "Repeat the last value" becomes a progressively worse assumption the further out the trend and seasonal cycle are expected to move. ARIMA's explicit trend/autoregressive structure has more room to demonstrate a genuine advantage once the naive baseline's flat-line assumption clearly breaks down.

Exercise 20.5 [E]. "Did you compare that 7.2 MAE against a naive baseline, just repeating the last observed value for all 28 days? This chapter's own measurement found exactly that comparison flipping the expected result: a naive baseline beat a reasonably-specified ARIMA model at this horizon. An MAE number in isolation, without the baseline comparison, doesn't tell you whether the model is adding value."

Exercise 20.6 [E]. Computing `.rolling(7).mean()` without `.shift(1)` first means the rolling window for row t includes row t's own value. The feature meant to summarise "the past 7 days" includes today's target value baked into its own input. That's a direct form of leakage: the model gets to see (an average containing) the answer as part of its own input feature.

Exercise 20.7 [R]. Discussion only. Hyndman and Athanasopoulos discuss measures like MASE (Mean Absolute Scaled Error), which scales errors relative to a naive/seasonal-naive baseline's own error rather than relative to the raw target values. It's designed specifically to allow comparison across series of very different scales, and to remain well-behaved when the series can be at or near zero, exactly the situation this chapter flags as breaking MAPE.

References

Beaulieu, Alan. 2005. *Learning SQL*. O'Reilly Media.

Bergstra, James, and Yoshua Bengio. 2012. "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research* 13: 281–305.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Box, George E. P., Gwilym M. Jenkins, Gregory C. Reinsel, and Greta M. Ljung. 2015. *Time Series Analysis: Forecasting and Control*. 5th ed. John Wiley & Sons.

Burkov, Andriy. 2019. *The Hundred-Page Machine Learning Book*. Andriy Burkov.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, CA), 785–94.

Chu-Carroll, Mark C. 2013. *Good Math: A Geek's Guide to the Beauty of Numbers, Logic, and Computation*. The Pragmatic Bookshelf.

Davis, Jesse, and Mark Goadrich. 2006. "The Relationship Between Precision-Recall and ROC Curves." *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, PA), 233–40.

Domingos, Pedro. 2012. "A Few Useful Things to Know about Machine Learning." *Communications of the ACM* 55 (10): 78–87.

Géron, Aurélien. 2019. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd ed. O'Reilly Media.

Holt, Charles C. 1957. *Forecasting Seasonals and Trends by Exponentially Weighted Moving Averages*. ONR Memorandum No. 52. Carnegie Institute of Technology.

Hughes, Arthur Middleton. 1994. *Strategic Database Marketing: The Masterplan for Starting and Managing a Profitable, Customer-Based Marketing Program*. Probus Publishing.

Humpherys, Jeffrey, Tyler J. Jarvis, and Emily J. Evans. 2017. *Foundations of Applied Mathematics, Volume 1: Mathematical Analysis*. Society for Industrial; Applied Mathematics.

Hyndman, Rob J., and George Athanasopoulos. 2018. *Forecasting: Principles and Practice*. 2nd ed. OTexts.

Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. 2008. "Isolation Forest." *2008 Eighth IEEE International Conference on Data Mining* (Pisa, Italy), 413–22.

Murphy, Kevin P. 2022. *Probabilistic Machine Learning: An Introduction*. MIT Press.

Murphy, Kevin P. 2023. *Probabilistic Machine Learning: Advanced Topics*. MIT Press.

Procida, Daniele. 2020. *Diataxis: A Systematic Framework for Technical Documentation*.

Robertson, Suzanne, and James Robertson. 2012. *Mastering the Requirements Process: Getting Requirements Right*. 3rd ed. Addison-Wesley.

Sayama, Hiroki. 2024. *Introduction to the Modeling and Analysis of Complex Systems*. LibreTexts.

Shannon, Claude E. 1948. "A Mathematical Theory of Communication." *Bell System Technical Journal* 27 (3): 379–423.

Thakur, Abhishek. 2020. *Approaching (Almost) Any Machine Learning Problem*. Abhishek Thakur.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems*.

Wickham, Hadley. n.d. *The Data Science Process*.

Winters, Peter R. 1960. "Forecasting Sales by Exponentially Weighted Moving Averages." *Management Science* 6 (3): 324–42.

Zheng, Alice, and Amanda Casari. 2018. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media.