Anyone Can Do Data Science

From tabletop to Python

Gregory Baker

Contents

Part I

The Visible Workflow

Chapter 1

Data Science Roles

Data science is rarely a one-person job. Someone has to frame the question, someone has to clean and organise the data, someone has to compare models, and someone has to keep the system reliable once other people depend on it.

1.1 Who usually does what?

The names vary from organisation to organisation, but the jobs are fairly recognisable.

Analysts. Analysts spend a lot of time turning messy operational data into tables, charts, and short explanations. They answer questions such as "Which stores are under-performing?" or "Did last month's promotion change sales?" A good analyst is not just a chart maker or storyteller. They are usually the first person to notice when the data are incomplete, mislabelled, or pointing to a different question than the one originally asked. In practice, analysts often work in Tableau or other business-intelligence tools because the job is partly explanatory as well as technical.

Data scientists. Data scientists tend to take over when the question needs modelling, experimentation, or forecasting. They might compare a few ways to predict churn, group stores into clusters, or test whether a new policy changes behaviour. In a course like this one, many of the methods we study sit in this part of the job.

Data engineers. Data engineers make sure the data arrive in a usable form and keep arriving tomorrow. They build and maintain the pipelines that collect, clean, store, and refresh information. If analysts and scientists are asking good questions but still fighting broken tables every week, engineering work is usually part of the answer.

Machine-learning engineers and adjacent roles. Some organisations split out a separate role for deploying and maintaining predictive systems. Other places fold that work into software engineering or data engineering. The exact title matters less than the

responsibility: once a model affects real decisions, someone has to monitor it, update it, and explain when it should not be trusted.

There is no perfect ladder through these roles. People move sideways as often as they move up. Someone might begin in analysis, learn enough modelling to move into data science, and then discover they enjoy systems work more than experiments. Another person may come from software and pick up the modelling later. The useful point is that the roles overlap, and the handovers matter.

1.2 How a project moves through a team

It helps to imagine a small, ordinary example rather than a grand AI story. Suppose a retailer wants to know where to open a new distribution point.

1. An analyst checks what data exist, whether store addresses are reliable, and which outcome actually matters: travel time, sales, spoilage, or all three.

2. A data scientist compares a few ways to group stores or predict demand and tests whether the patterns are stable enough to inform a decision.

3. A data engineer turns the messy one-off extract into something repeatable so the analysis can be refreshed rather than rebuilt from scratch.

4. A delivery or product team decides how much of the result should influence the real business decision and what human review is still needed.

It is careful, cumulative work. The team keeps asking simple questions: Do we trust the data? Does the model beat a sensible baseline? Will the result still make sense when the context changes?

1.3 Where we might fit

We do not need to love every part of the pipeline. Some people enjoy turning vague questions into clean tables. Others like the modelling. Others prefer building reliable systems around the analysis. Good teams need all three kinds of work.

Domain knowledge matters too. Someone who understands logistics, health, finance, design, or public policy often asks better data questions than someone with stronger coding but weaker context. Business students do not need to become faux computer scientists overnight. Computing students do not need to pretend that implementation is the whole story. The point is to build enough shared language that both groups can work on the same problem honestly.

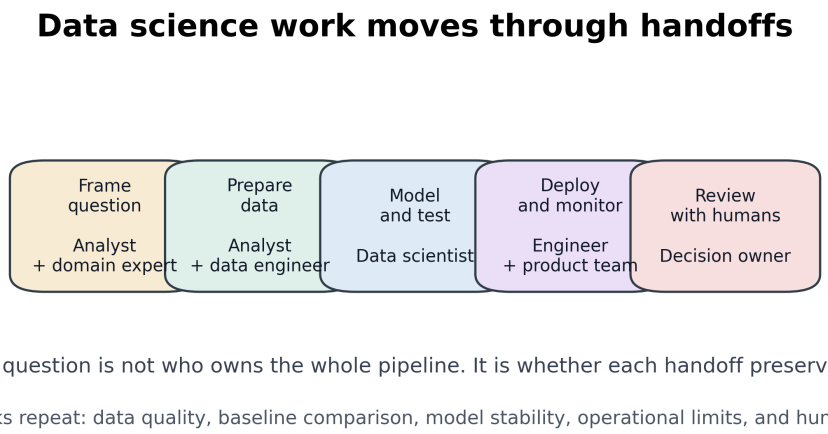


Figure 1.1: A small data-science project moves through repeated handoffs. The important discipline is preserving context as work passes from question framing to preparation, modelling, deployment, and human review.

Chapter exercises

1. Read the following project brief: a retailer wants to know why some stores miss delivery windows. List the analyst, data scientist, data engineer, machine-learning engineer, and domain-expert contributions that would make the project credible.

2. Sketch the handoff path for a customer-churn project from the first stakeholder question through monitoring after deployment. Mark one place where context could easily be lost.

3. Rewrite the vague request "Can we use AI to improve sales?" as three answerable data questions. For each question, name one dataset or measurement you would need.

4. Choose two roles from this chapter that suit different strengths. For each role, describe one portfolio artefact a reader could create during this course.

Chapter 2

Tableau for Rapid Visual Exploration

Chapter overview

Business intelligence tools accelerate exploratory visualisation when structured data arrives and stakeholders need charts immediately. We introduce Tableau's drag-and-drop workflow—from importing CSV files to assembling multi-chart dashboards—and discuss when reaching for a BI platform beats writing code. The chapter concludes with export strategies so any Tableau figure can travel into slide decks or reports.

2.1 Understand when to use business intelligence tools

Choose the right platform by weighing speed, reproducibility, and the composition of the analytics team.

Learning objectives

After this section we will be able to:

- articulate the core purpose of business intelligence platforms such as Tableau, Power BI, Qlik, and Amazon QuickSight;
- describe the typical ratio of data analysts to data scientists in most organisations and explain how infrastructure reflects that headcount;
- decide whether a BI tool or a code-driven workflow better fits a given exploratory question.

Prerequisites

We should already be comfortable with:

- reading rows and columns in a spreadsheet or CSV file;
- interpreting scatter plots, bar charts, and line charts;

- basic discussion of role differences across analytics teams.

Business intelligence tools prioritise speed: a polished bar chart that might take two or three iterations in code materialises in minutes through drag-and-drop interactions. That acceleration matters when a stakeholder asks for a last-minute update before a presentation or when the same dataset needs multiple lenses—revenue by quarter, sales by region, customer segments by churn—each requiring minimal code literacy.

In most organisations data analysts outnumber data scientists by a wide margin, and those analysts spend the majority of their time creating visualisations for decision-makers. The infrastructure reflects that headcount: centrally hosted Tableau or Power BI servers offer pre-approved data connections, role-based publishing permissions, and scheduled refreshes that keep dashboards current. Data scientists still lean on Python or R when models need custom feature engineering, when reproducibility demands version-controlled notebooks, or when the analysis pipeline will run in production. Yet knowing how to spin up a Tableau worksheet means we can generate an exploratory chart in minutes rather than hours when time is tight.

Popular platforms include:

- **Power BI.** Tightly integrated with the Microsoft Office ecosystem; frequently bundled with enterprise Windows licenses.
- **Tableau.** Long-established visual-analytics platform with strong market presence, though licensing costs can be high.
- **Qlik.** Common in financial-services environments; emphasises associative data models.
- **Amazon QuickSight.** Cloud-native BI service popular among startups and AWS-heavy organisations.

All four share a common pattern: import structured data, assign columns to visual channels—position, colour, size—and publish the resulting dashboards to a web portal where colleagues can filter or drill down without writing queries.

Decision matrix. When structured data already sits in a database or flat file and the audience expects interactive charts within the hour, a BI tool often wins. When the workflow requires custom transformations, machine learning pipelines, or version-controlled reproducibility for peer review, code-driven Python or R workflows remain essential. Recognise both contexts so we pick the tool that matches the deadline and the team's skill mix.

Section summary

- Business intelligence platforms accelerate chart production when structured data is ready and stakeholders need results fast.

- Data analysts typically outnumber data scientists; BI infrastructure reflects that composition.
- Choose between BI drag-and-drop and code-driven workflows by weighing time constraints, reproducibility requirements, and team capabilities.

Self-check questions

1. Under what conditions would we recommend Tableau over a Python notebook to a colleague?
   Hint: consider deadline pressure, data readiness, and the recipient's technical background.

2. How might the wide ratio of analysts to data scientists influence an organisation's technology roadmap?
   Hint: think about where training budgets, server licenses, and governance policies flow.

2.2 Connect data sources in Tableau

Import CSV files, inspect field types, and navigate to worksheets to begin plotting.

Learning objectives

After this section we will be able to:

- open a CSV or Excel file in Tableau and review the automatically detected column types;
- verify that numeric, text, and date columns match our expectations;
- move from the data-source screen to a blank worksheet where visualisations begin.

Prerequisites

Make sure we can:

- locate a CSV or Excel file on the local file system or a shared drive;
- recognise basic data types (strings, integers, floating-point numbers, dates);
- describe what it means to import a CSV and inspect its columns before plotting.

Starting Tableau presents a connection menu listing text files, Excel workbooks, database servers, and cloud connectors. Selecting a CSV immediately displays a grid preview similar to opening a structured table in a spreadsheet or notebook. Tableau inspects the first few rows to guess whether each column holds text, whole numbers, decimals, or dates, and it highlights any ambiguous cases with a warning icon.

For example, a column named `Year` containing `2015, 2016, 2017` defaults to a numeric type, while `Brand` with entries such as `Toyota, Honda, Ford` becomes text. A `Date` field showing `2024-08-15` triggers date recognition so Tableau can later aggregate by month or quarter. If Tableau misclassifies a column—treating a numeric product code as a measure instead of a categorical dimension—we click the type icon to override the inference before proceeding.

Once the preview looks correct, clicking the worksheet tab at the bottom spawns a blank canvas with two main areas: a left-hand pane listing all detected fields organised into *dimensions* (categorical) and *measures* (numeric), and a central grid marked *Columns* and *Rows* where we drag those fields to build charts. If we later rebuild the same analysis in code, this plays the role of an import step plus a plotting call; here all those steps happen through mouse actions rather than method calls.

Troubleshooting clinic. If the file fails to load, check for mismatched delimiters (comma versus semicolon) or unexpected encoding. Tableau guesses UTF-8 by default; older exports sometimes require ISO-8859-1. If a date column shows as text, ensure the format matches a recognised pattern such as `YYYY-MM-DD` or `DD/MM/YYYY`. Correcting the type at import prevents downstream aggregation errors.

Section summary

- Tableau imports CSV and Excel files with automatic type detection.
- Review the dimension versus measure classification before building worksheets.
- Navigate to the worksheet tab to start dragging fields into visual encodings.

Self-check questions

1. How does Tableau's dimension versus measure split compare with categorical versus numeric columns in a dataset?
   Hint: consider which operations make sense for each type.

2. When might we override Tableau's automatic type detection, and what signals would we look for?
   Hint: think about ID codes, postal codes, or year fields that should not be summed.

2.3 Build worksheets with drag-and-drop encodings

Create bar charts, line charts, and scatter plots by assigning columns to position, colour, and size channels.

Learning objectives

After this section we will be able to:

- drag dimensions onto the Columns or Rows shelf to define chart axes;
- assign measures to aggregation functions (count, sum, average, median) and understand when each suits the question;
- apply sorting, filtering, and trend lines to reveal patterns in the data;
- switch between chart types (bar, line, scatter) by selecting marks from the palette.

Prerequisites

We should already know how to:

- interpret aggregated statistics (mean, median, count);
- distinguish between categorical and numeric variables;
- describe how scatter plots differ from bar charts in terms of what they reveal.

2.3.1 Bar chart: vehicle body types

Start by answering "How many listings does each body type have?" Drag the `Body Type` dimension to the Columns shelf; Tableau groups the data by that categorical field. Then drag any measure—say, the count of rows—to the Rows shelf so Tableau tallies how many records belong to each body type. The default chart may appear as vertical bars; clicking the horizontal bar icon in the marks palette rotates them into a side-by-side layout that reads like a league table.

The initial ordering might be alphabetical, which obscures which body type dominates. Right-click the axis, choose *Sort*, and select descending order by the row count. SUVs rise to the top, followed by hatchbacks, utes, and smaller categories trailing off toward zero. Sorting transforms a jumbled list into a narrative: "SUVs lead the market, hatchbacks hold second place, and niche body types cluster at the low end."

Renaming the measure to "Number of vehicles by body type" clarifies the axis label so stakeholders reading the exported chart immediately grasp what the bars represent. That small edit prevents confusion when the worksheet appears in a dashboard alongside other counts.

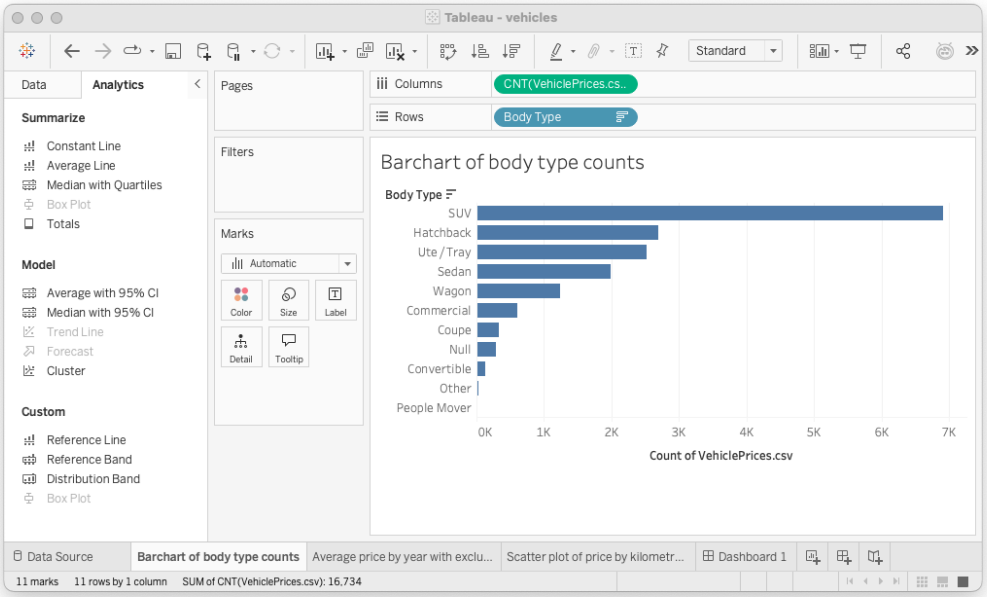


Figure 2.1: The body-type worksheet after sorting the counts into descending order.

2.3.2 Line chart: average price over time

Create a new worksheet to explore how average listing prices change with vehicle age. Drag `Year` to the Columns shelf so time flows left to right, then place `Price` on the Rows shelf. Tableau defaults to summing prices, which inflates the y-axis nonsensically. Click the `Price` pill, switch the aggregation to *Average*, and the chart recalibrates to show mean prices per year.

Adding `Brand` to the *Detail* shelf or *Tooltip* shelf layers in vehicle makes; hovering over a spike reveals it's a 1959 Ferrari priced at $1.5 million. That outlier skews the average dramatically. We can right-click the data point and exclude it, or we can switch the aggregation from average to *Median* so extreme values no longer dominate the summary.

The resulting trend shows prices declining until around the year 2000, then creeping back upward as older cars enter the vintage market—a phenomenon easier to spot once the y-axis stabilises. Filtering years to only those with at least ten listings removes singleton spikes and focuses the narrative on well-populated cohorts.

2.3.3 Scatter plot: price versus kilometres with trend lines

A scatter plot asking "Do higher-mileage vehicles cost less?" requires numeric axes for both dimensions. Drag `Kilometres` to Columns and `Price` to Rows. Tableau may aggregate both fields by default, collapsing the scatter into a single point. Right-click each pill and convert it to a *Dimension* so every row plots individually.

The raw scatter suffers from overplotting near the origin because most listings cluster at low mileage and modest prices. Editing the y-axis to use a logarithmic scale spreads the dots vertically, revealing structure. Dragging `Condition` (new, ex-demo, used) onto the

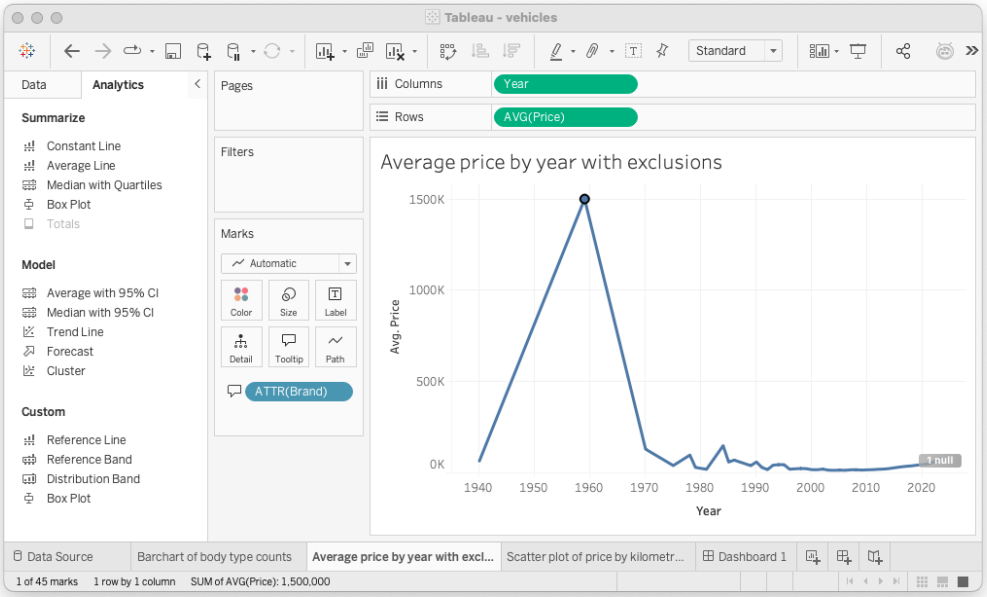


Figure 2.2: The line-chart worksheet makes the outlier year visible before we decide whether to exclude it or switch to a more robust summary.

Color shelf splits the cloud into three bands: new cars hug the left edge at zero kilometres, demos span a narrow middle range, and used vehicles trail downward as mileage climbs.

Converting the x-axis to logarithmic further clarifies the separation. Now we can add a trend line: open the *Analytics* pane, drag *Trend Line* onto the chart, and choose a model type. On these log–log axes a power law appears as a straight line, so a power-law model often fits price against kilometres better than a straight-line fit on the raw scales. Tableau overlays a separate trend for each condition category, showing that used cars exhibit a strong negative relationship (more kilometres, lower price), while new cars show a weak, possibly spurious upward trend—hovering over that line reveals a tiny R^2 value and a large p-value, suggesting noise rather than signal.

The used-car trend, by contrast, shows a much higher R^2 and a small p-value, confirming the intuitive relationship between wear and depreciation. Rename the worksheet to “Price versus usage” so colleagues understand the narrative at a glance.

Accessibility spotlight. When scatter plots use colour to distinguish groups, supplement the encoding with shape or size if the chart will be printed in greyscale or viewed by individuals with colour-vision deficiency. Tableau’s shape palette provides triangles, circles, and squares that remain distinct without hue.

Section summary

- Drag dimensions and measures onto Columns, Rows, Color, Size, and Detail shelves to encode data visually.

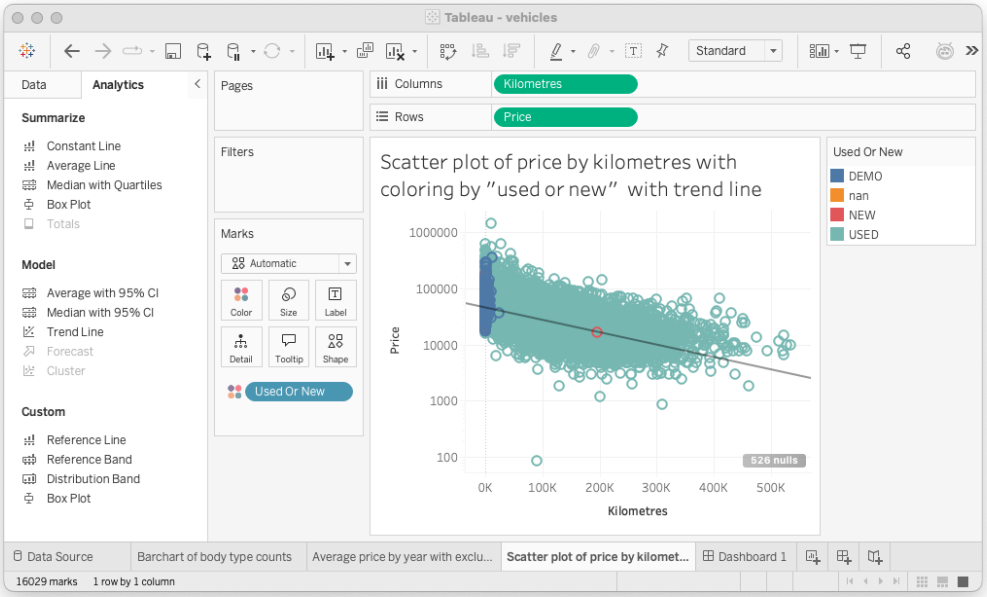


Figure 2.3: The price-versus-kilometres scatter plot with colour coding and a fitted trend line.

- Sort, filter, and switch aggregation functions (count, average, median) to clarify patterns and remove noise.
- Apply trend lines from the Analytics pane; inspect fit statistics carefully and treat them as supporting evidence rather than automatic proof.
- Use logarithmic axes when data span multiple orders of magnitude.

Self-check questions

1. When would we choose median over mean for a price aggregation, and how does that choice affect interpretation?
   Hint: consider outlier sensitivity and the story we want the chart to tell.

2. How does converting a measure to a dimension change the way Tableau plots the data?
   Hint: think about whether values get aggregated or plotted individually.

3. What does it suggest when a trend line shows a tiny R^2 value and only weak statistical support?
   Hint: ask whether the apparent relationship is strong enough to guide a decision.

2.4 Assemble dashboards for stakeholders

Combine multiple worksheets into a unified view that updates automatically when data refreshes.

Learning objectives

After this section we will be able to:

- create a new dashboard and drag worksheets onto a grid layout;
- resize and reposition charts so related insights sit side by side;
- explain how publishing dashboards to a Tableau Server or Tableau Public enables organisation-wide access.

Prerequisites

We should have:

- completed at least two worksheets with distinct questions (for example, a bar chart and a scatter plot);
- basic familiarity with arranging multiple charts on a single page.

Click the dashboard tab at the bottom of the Tableau window to open a blank canvas. The left pane lists all available worksheets; dragging one onto the canvas anchors it in place. Drag the price-over-time line chart into the top half, treating it as a wide banner that emphasises temporal trends. Place the body-type bar chart in the bottom-left quadrant as a compact summary, then drop the price-versus-usage scatter into the bottom-right corner.

Tableau reflows the worksheets automatically, adjusting axis labels and titles to fit the allocated space. If new data arrives—for instance, the CSV gets updated with this week's listings—refreshing the data source propagates changes to all three charts simultaneously. That live-update property makes dashboards valuable for operational monitoring: sales teams can check daily revenue, support analysts can track ticket volumes, and logistics coordinators can review dispatch times, all without re-running code.

Publishing the dashboard to a Tableau Server or Tableau Public renders it accessible via a web URL. Colleagues with appropriate permissions can filter, zoom, and export individual charts without opening Tableau Desktop. That browser-based interactivity bridges the gap between data analysts who build the visualisations and decision-makers who consume them.

Deployment note. Tableau Server requires enterprise licensing and IT setup; Tableau Public is free but makes all dashboards publicly visible on the internet. Choose the deployment model that matches the sensitivity of the data and the organisation's governance policies.

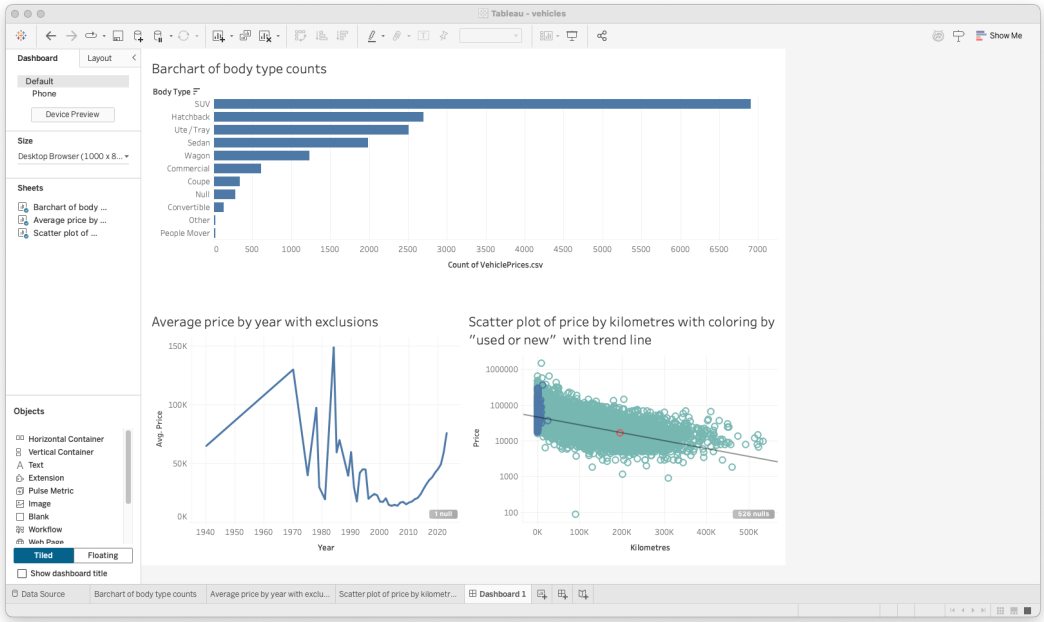


Figure 2.4: The dashboard combines the body-type counts, the year-by-price worksheet, and the price-versus-kilometres scatter into one stakeholder-facing page.

Section summary

- Drag worksheets onto a dashboard canvas to unify related charts in a single view.
- Tableau automatically propagates data refreshes across all dashboard components.
- Publish to a server or public portal so stakeholders can explore the data through a browser without Tableau Desktop.

Self-check questions

1. What advantage does a unified dashboard offer over emailing individual chart images to stakeholders?
   Hint: consider interactivity, version control, and automatic updates.

2. How would we decide between Tableau Server and Tableau Public when publishing a dashboard?
   Hint: weigh data confidentiality against cost and access requirements.

2.5 Export images for reports and slide decks

Save individual worksheets or entire dashboards as high-resolution PNG or vector PDF files.

Learning objectives

After this section we will be able to:

- export a worksheet or dashboard via the *Worksheet* → *Export* menu;
- choose between PNG (raster) and PDF (vector) formats based on downstream usage;
- control image resolution so printed materials remain sharp;
- integrate exported Tableau figures into LaTeX documents, slide presentations, and web articles.

Prerequisites

Make sure we understand:

- the difference between raster and vector graphics;
- how resolution (DPI) affects print quality;
- basic file-system navigation to locate saved exports.

Tableau embeds charts inline when we work in Desktop, but stakeholders often need static images for slide decks, PDF reports, or printed handouts. Right-click a worksheet tab (or select *Worksheet* → *Export* → *Image* from the menu) to save a PNG snapshot. Tableau defaults to screen resolution, which suffices for digital slides; increasing the DPI to 300 in the export dialog ensures the text and markers stay crisp on paper.

When the destination supports vector graphics—LaTeX via `\includegraphics`, Adobe Illustrator, or Inkscape—choose PDF export so axes, labels, and trend lines remain editable and scale without pixelation. Dashboards export the same way: right-click the dashboard tab and select *Export Image* to capture all constituent worksheets in a single file. That unified export mirrors the multi-panel figures we later build in chapter 24, preserving layout and alignment.

Tableau also supports programmatic export through its REST API when batch image generation is required—for example, nightly snapshots of operational dashboards emailed to managers. Scripting that workflow falls outside exploratory analysis but becomes relevant once dashboards move into production monitoring.

Filename discipline. Use descriptive names that state the worksheet, metric, and date rather than generic labels such as `figure1.png`. Store exports in a dedicated `figures/` directory alongside the LaTeX source or slide deck to simplify path management.

Section summary

- Export worksheets and dashboards as PNG for digital slides or PDF for print and vector editing.
- Increase DPI to 300 when the figure will be printed or included in formal publications.
- Adopt consistent filename conventions so the exported images integrate cleanly into reports and version-controlled repositories.

Self-check questions

1. Under what conditions would we choose PDF export over PNG when saving a Tableau worksheet?
   Hint: consider whether the recipient might need to edit text labels or scale the figure.

2. How does Tableau's image-export workflow compare with exporting a figure from code regarding resolution and format control?
   Hint: think about the choices each tool exposes and how they map to print versus screen requirements.

Concluding bridge

Tableau excels when structured data already exists, stakeholders need results within the hour, and the team includes analysts who prefer visual interfaces to code. The drag-and-drop workflow—connecting data, building worksheets, assembling dashboards, and exporting images—mirrors the same analytic habits we use elsewhere in the course, except that each step happens through mouse actions rather than method calls.

The next chapters return to the low-code Orange workflow that anchors the main path. Later, the extension part revisits code-driven visualisation in chapter 24 and chapter 25. Treat Tableau and code as complementary tools: reach for Tableau when speed and stakeholder interactivity dominate, and lean on code when reproducibility, version control, and custom transformations matter most.

Chapter exercises

1. Open a small vehicle or sales CSV in Tableau. Record three fields Tableau treats as dimensions, three it treats as measures, and one field whose detected type you would check manually.
2. Build three worksheets from the same dataset: a count bar chart, an average-over-time line chart, and a scatter plot with a trend line. Write one sentence explaining the question each worksheet answers.

3. Assemble those worksheets into a dashboard for a manager who has five minutes before a meeting. Explain the layout choices and which chart should draw attention first.

4. Export one worksheet for a printed report and one for a slide deck. Choose PNG or PDF deliberately and propose filenames that would still make sense next semester.

5. Identify one point in the workflow where Tableau is the better tool, and one point where a Python notebook would become the safer choice.

Chapter 3

Introduction to Orange

Chapter overview

The textbook deliberately uses three tool families. Orange keeps the workflow visible. Tableau is strongest when the job is visualisation. Python comes later when we need reproducible code, automation, or libraries outside Orange. This chapter stays with the visual tools and turns Orange into the main working environment for the early part of the book.

That ordering is deliberate. A visual canvas lets beginners see the modelling logic before syntax becomes one more thing to fight with. By the time we shift into Python later in the textbook, the steps should already feel familiar: load a table, choose columns, preprocess the data, fit a learner, and inspect what came out. That is a gentler transition than dropping raw code into the opening weeks and hoping everyone keeps up.

3.1 Orange

Orange is a visual data-mining application built around a **canvas** of connected **widgets**. A widget is a small tool with one job: load data, draw a plot, fit a learner, or summarise results. When we join widgets together, we get a **workflow**. That matters in an introductory course because the logic stays visible. We can literally point to the sequence: load the table, choose the target, fit the model, inspect the output.

Readers new to programming can focus on the statistical idea first. We can ask sensible questions about features, targets, train/test splits, clusters, or probabilities before worrying about syntax.

Tableau in one sentence. Tableau is excellent when the task is mainly visualisation and dashboarding. Orange is the better default for this textbook because it stays closer to the modelling workflow as well as the plots.

3.2 What each saved file means

This distinction causes confusion early on, so it is worth making explicit.

- An Orange `.ows` file stores the **workflow**: which widgets we used, how they are connected, and the settings we saved on the canvas.
- A dataset export stores the **data**: for example a cleaned CSV or tab-delimited table produced after filtering, selecting columns, or creating new features.
- A saved model stores the **trained result**: the fitted learner rather than the whole workflow.

So when we say "save the workflow", we mean the Orange canvas. When we say "export the cleaned data", we mean a table that another tool can read. Those are related, but they are not the same artefact.

3.3 Common language

Term	Meaning in this textbook
Feature	An input column used to help make a prediction or discover structure.
Target	The column we want to predict in a supervised task.
Class	A categorical target, such as "yes" or "no".
Learner	The fitting procedure before training, such as logistic regression or k-NN.
Model	The trained result produced after the learner sees data.
Workflow	A connected Orange canvas showing the analysis steps.
Pipeline	A code-based sequence of preprocessing and modelling steps, usually in scikit-learn.
Validation	The evidence we use while choosing settings or comparing models.
Test set	A final held-out slice used only after we stop tuning.
Baseline	A simple reference method that keeps more complex models honest.
Calibration	Whether predicted probabilities match observed frequencies.

3.4 Stage Data for Orange Workflows

Lay out a reliable canvas by importing clean tables, confirming data types, and annotating each widget before deeper exploration begins.

Learning objectives

After reading this section, we will be able to:

- configure the **File**, **Select Columns**, and **Data Table** widgets so the entire canvas shares labelled, trustworthy features;

- profile CSV columns to spot missing values, mixed data types, and outliers before wiring analysis widgets;
- document decisions on the canvas so later modelling inherits the exploratory context.

Beginner bridge: reading the Orange interface Orange arranges controls around the canvas so we can operate entirely with the keyboard if needed. Press `Ctrl+O` to open datasets, `Ctrl+S` to save workflows, and tap the space bar to open the widget search without reaching for the mouse. Focus moves between widgets with the arrow keys, while `Ctrl+=` and `Ctrl+-` zoom the canvas for low-vision readers. In *Options* → *Preferences*, the *General* tab lets us enlarge fonts and switch to the high-contrast icon set.

Staging begins by dragging a **File** widget onto a new canvas and pointing it at a structured dataset such as the Sydney housing CSV. We tick the options to detect variable types automatically and to guess special roles. The data preview lets us confirm that sale prices, lot sizes, and suburb indicators are recognised correctly before anything flows downstream. When columns misbehave—postcode encoded as text, or distance to the central business district stored as a string—we fix them immediately in **Select Columns**. The widget exposes each feature, allowing us to promote identifiers to meta fields, drop redundant columns, and rename fields so later charts read cleanly.

Next we add a **Data Table** widget to inspect sampled rows. Sorting by price exposes extreme transactions that may warrant closer attention in the regression chapter, while the summary bar highlights missing values. We annotate the widget with concise notes such as "postcode promoted to meta" or "distance parsed as numeric". These annotations become the canvas-level documentation other readers will rely on later.

A simple staging strip usually looks like this: **File** → **Select Columns** → **Data Table**, with short annotations explaining any type corrections or dropped fields.

Step-by-step: create a reusable staging template.

1. Place **File**, **Select Columns**, and **Data Table** widgets in a horizontal line.
2. Configure **File** to load the housing CSV with "First row as header" and "Auto detect types" enabled.
3. In **Select Columns**, drag geographic descriptors (suburb, postcode) into meta, retain numerical predictors (price, distance, rooms) as features, and set the target to sale price.
4. Preview the transformed table, filtering for recent sale years to ensure the date parsing succeeded.
5. If you want a reusable cleaned table, add **Save Data** and export the filtered data as a tabular file. Save the Orange workflow itself separately as an `.ows` file.

Try this variation. Swap in the Palmer Penguins dataset from the supplied practice bundle and repeat the staging steps. Map species to meta fields, treat flipper length as a numeric feature, and note how Orange automatically recognises the categorical island variable.

Troubleshooting. If the **File** widget shows question marks instead of preview data, check that the CSV path does not include accented characters—older Orange versions prefer ASCII file names. When columns load with the wrong type, toggle "Auto detect types" off, set the correct role manually, and add a note to the widget so collaborators know why the override was necessary.

> **Computing extension.** Script the same staging steps with Orange's Python Script widget or a separate notebook when you want a versioned preprocessing pipeline alongside the GUI.

3.5 See Patterns Through Interactive Visuals

Use Orange visualisations to surface structure in real data, switching encodings until relationships become obvious.

Learning objectives

After reading this section, we will be able to:

- configure **Scatter Plot**, **Box Plot**, and **Distributions** widgets to compare subgroups and time periods;
- layer colour and size channels to expose structure in the data;
- compose **Scatter Plot** and **Linear Regression** widgets into a diagnostic loop for identifying nonlinear effects.

We begin by connecting a **Scatter Plot** widget to the staging pipeline. Orange automatically proposes axes; we set lot size on the horizontal axis and sale price on the vertical axis, then enable jitter to separate overlapping points. Colouring by dwelling type reveals that detached houses occupy larger parcels, with terraces and apartments clustered toward smaller footprints. Switching the colour variable to distance from the central business district exposes a second pattern: outer-ring properties often fall into lower price bands than inner-city dwellings even when lot sizes match. Size encoding adds a third dimension by making bedroom counts visible.

To compare suburbs, we drop a **Box Plot** widget and select the suburb field as the grouping variable. Sorting by median price shows which regions command a premium. Hovering over each box lists interquartile ranges and counts, helping us identify suburbs with thin sales volume that may be too noisy for modelling. When prices shift across years,

the **Line Plot** widget summarises rolling averages and signals when structural market changes need to become explicit features later.

Visual checklist for every dataset.

1. Plot one plausible predictor against the main outcome, then change the colour field at least twice.
2. Add a box plot or distribution plot to compare subgroups instead of relying on one scatter plot alone.
3. Inspect time if time exists; many surprises are really time effects rather than feature effects.
4. Save one useful selection so it can travel into later modelling experiments.

Orange's interactivity invites exploration. Selecting a cluster in the scatter plot highlights the same rows in the **Data Table** and filters the **Box Plot**. That makes it easy to drill into "inner-west terraces under $1 million" or "outer suburban apartments post-2019" and inspect the matching records immediately.

A first real clustering dataset. Once the basic workflow feels comfortable, switch to the "Grades for English and Math" dataset. Feed it through **k-Means**, open **Scatter Plot**, and then add **Silhouette Plot**. It is a small step up from toy points on the canvas, but it already feels like real unsupervised learning rather than a pure demonstration.

Troubleshooting. If linked selections stop synchronising, confirm that "Send selections" remains enabled in each widget's control panel. For colour palettes that fail accessibility checks, switch to a colour-blind-friendly palette so hue differences remain legible when printed in grayscale.

3.6 Link Exploration to the Broader Workflow

Anchor visual discoveries within an end-to-end checklist so exploration flows naturally into modelling, evaluation, and communication.

Learning objectives

After reading this section, we will be able to:

- map each stage of a data-science checklist to specific Orange widgets;
- design canvases that separate exploratory, modelling, and reporting zones for clarity;
- export evidence packs that make later code-based work more efficient.

The checklist begins with **preamble** and **fetch**: documented in the staging strip by noting data provenance and cleaning choices. **Visualise** and **clean** correspond to the scatter, box, and line plots we assembled, plus any feature filters or imputation widgets we add when missing data appears. When we reach **engineer**, we branch the canvas so we can experiment with **Formula** or other transformation widgets without disturbing the earlier steps.

Splitting data for modelling is explicit: we chain **Select Columns** → **Data Sampler** → modelling widgets and finish with **Test & Score**. Evaluation metrics travel into **Report** widgets, where we attach interpretations describing why one model outran another. Orange's **Rank** widget helps justify the "choose" stage, while exported PNGs or PDFs capture the evidence we may later recreate in Python.

Finally, we treat the canvas as a living notebook. Each branch is labelled with the stage it serves. When new data arrives, we reopen the `.ows` file, refresh the **File** widget, and retest assumptions. Exported selections and cleaned tables then feed the later hands-on chapters and give code-based workflows a reliable starting point.

Try this variation. Branch the staged data into a classification pipeline—add **Logistic Regression** and **Confusion Matrix** widgets to predict whether a listing sells within 30 days, or use penguin species to demonstrate the same idea on a smaller dataset.

Troubleshooting. If **Data Sampler** refuses to split the data as expected, double-check that the target variable is marked as "class" or numeric as appropriate in **Select Columns**. When exports omit annotations, enable "Embed annotations" in the **Report** widget options so the canvas commentary survives in the PDF.

Chapter exercises

1. Match each artefact to the question it can answer: an Orange `.ows` workflow, a cleaned CSV export, a saved model file, a screenshot, and a report export.

2. Build a staging strip for the housing sample using **File**, **Select Columns**, and **Data Table**. Add a short canvas note explaining one type or role decision.

3. Add at least two visual widgets to the staged data. Record one visible pattern, one possible data-quality issue, and one follow-up question.

4. Branch the staged data into a simple modelling workflow with a learner, **Data Sampler**, **Test & Score**, and **Report**. Explain what changed when exploration became modelling.

Chapter 4

Hands-on Activity 1: k-Means by Hand

This session is all about grouping similar things together by eye before we let software take over. We sort purple counters into clusters, notice which tokens naturally travel together, and then rebuild the same idea inside Orange so the computer follows our reasoning.

4.1 Prepare the Materials

Each group needs a small kit so everyone can participate in the physical clustering exercise before opening Orange.

- **Purple counters or tokens.** These are the data points.
- **A few movable centroid markers.** Small dinosaur figures or contrasting counters work well.
- **A ruler.** This helps us judge distances during the assignment step.
- **A phone or camera.** Take one photograph of the finished layout before rebuilding it digitally.

4.2 Cluster Tokens by Hand

Gather everyone around a table with their purple counters, centroid markers, and a ruler.

1. Scatter the purple tokens across the table. Create a few tight groups and leave some points more isolated.
2. Start with three clusters. Place three centroid markers anywhere on the table.
3. For each purple token, decide which centroid is closest. Use the ruler if you need help judging distances, and mark the token's temporary assignment with a matching coloured counter if you have one.

Figure 4.1: A minimal clustering kit for the opening k-means activity: purple counters, centroid markers, and a ruler for distance checks during the assign step.

Figure 4.2: Coloured counters arranged on a tabletop with dinosaur figurines ready to act as centroids, showing how physical tokens can represent data points during clustering.

4. Move each centroid to the mean position of the purple tokens that chose it. A grid and precise coordinates would give a perfect mean, but a careful visual estimate works for this demonstration.

5. If a centroid is lonely, relocate it to the most distant purple token.

6. Repeat the assign–update cycle until the centroids stop moving. Take a photograph of the finished layout.

Try a few variations once you have a stable configuration:

- Reset the centroids in different starting positions. The clusters should settle in similar places even if the labels change.
- Begin with two centroids very close together and observe how long it takes for them to drift apart.
- Place one centroid far from the data to see the lonely-centroid scenario in action.
- Experiment with four or five centroids if you have enough colours.

4.3 Recreate the Workflow in Orange

Now translate the physical intuition into a digital workflow.

1. Open Orange and drag a **Paint Data** widget from the Data palette onto the canvas.

2. Double-click the widget and sketch the configuration from your photograph. Precision is not essential; the aim is to reflect the broad structure of your tabletop clusters.

3. Add the **k-Means** widget from the Unsupervised palette and connect the painted data to it. Fix the number of clusters to match the number of centroids you used.

4. Drop a **Scatter Plot** widget onto the canvas, connect the output of k-Means, open the plot, and colour points by cluster. Compare it with your photograph.

5. Keep the scatter plot open while trying different numbers of clusters in the k-Means widget. Note how the assignments change.

6. Give k-Means a range of cluster counts so it can suggest the value that maximises the silhouette score.

7. Add a **Silhouette Plot**. Connect the output of k-Means to it, and forward the silhouette scores to the scatter plot so everyone can see which points feel confidently placed and which sit on the fence.

8. In the silhouette plot, select a point with a high score and inspect its position in the scatter plot. Repeat for a point with a low score.

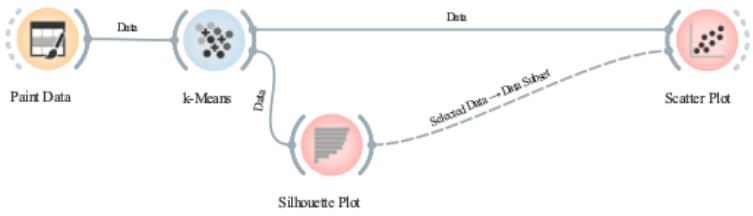


Figure 4.3: Orange workflow screenshot with Paint Data feeding into k-Means, Scatter Plot, and Silhouette Plot widgets, illustrating the digital mirror of the tabletop activity.

9. Save the workflow for later reference.

The next chapter uses the same k-means logic on a business dataset with real store locations and a practical decision to support.

Chapter exercises

1. Before opening Orange, record your group's tabletop clusters. Note the number of clusters, the reason for that choice, and one point that felt ambiguous.
2. Recreate the points with **Paint Data**. Run k-means with two different values of k, save a workflow screenshot, and compare both software results with the tabletop grouping.
3. Open **Silhouette Plot**. Does it support the value of k your group chose by eye? Explain your answer using the plot rather than just the cluster colours.
4. Imagine stretching the horizontal axis so distances in that direction count twice as much. Which points would be most likely to change clusters, and why?

Chapter 5

Retail Analytics Case Study: Mapping Store Clusters

Chapter overview

We take the clustering intuition from the k-means chapter and translate it into a concrete business problem. The chapter starts with the raw data we actually have, shows how geography, scaling, and missing values shape the workflow, and then uses clustering to support a depot-planning recommendation.

5.1 Business Context and Data Assets

We start with the business problem and the imperfect spreadsheet we inherit.

Picture us as the analytics crew for a wholesaler that wants more reach without blowing the logistics budget. The team has a spreadsheet of independent grocery stores that specialise in Indian produce. All we have at the start is a list of addresses, local areas, and rough turnover estimates.

Our immediate question is simple: how many distribution depots should we run across the metro area? Depots cost real money, but marathon delivery runs spoil fresh produce and wear out store managers' patience. Clustering the stores into tidy territories lets each depot look after a compact region.

Store	Suburb	Latitude	Longitude	Turnover band
Harris Park Pantry	Harris Park	-33.822	150.989	Medium
Auburn Fresh Foods	Auburn	-33.850	151.033	High
Liverpool Spice Market	Liverpool	-33.920	150.923	Medium
Blacktown Grocer	Blacktown	-33.768	150.906	Low

Table 5.1: A tiny representative extract showing the kind of fields we begin with before geocoding checks and feature engineering.

International adaptation guide. Swap in local chains or community grocers if you are teaching outside Australia. Replace suburb names, supplier references,

and demographic context with local equivalents drawn from the region you serve. The analytical flow stays the same while the story feels authentic to your cohort.

5.2 Feature Engineering and Visual Exploration

We transform the raw spreadsheet into spatial features and visual diagnostics that inform clustering choices.

We start by geocoding each store address. In practice that means sending the addresses to a service such as OpenStreetMap Nominatim or Google Maps and saving the returned latitude and longitude. We then manually check the obvious failures. Real geocoding work is messy: some addresses are partial, some stores sit inside shopping centres, and some rows need human correction before the map is trustworthy. That is why the first pass is always part technical and part clerical.

Once those coordinates exist, we load the table into Orange and throw the points onto a scatter plot or map. Longitude on the x-axis and latitude on the y-axis already sketch the city, showing chains of stores hugging train lines and main roads. That visual map is the first reality check: before we run any clustering, we want to know whether the addresses look plausible and whether the spread of stores matches the business story.

> **Geography caution.** Latitude and longitude are angular coordinates, not linear distances. The extreme cases make that obvious: near the poles, one degree of longitude covers almost no ground at all, while one degree of latitude still covers a large north–south distance. Around Sydney, a practical classroom fix is to replace longitude λ with $\lambda_{\text{adj}} = \lambda \cos(\varphi)$, where φ is the local latitude, before measuring straight-line distances. It is still an approximation, but it is an honest one.

On real data, this is also the point where we stop pretending the dataset is complete. Store addresses may be current while turnover estimates are stale. A store can look geographically isolated yet still be commercially tied to the rest of the network through supplier contracts, warehouse access, or delivery windows. So the feature-engineering pass is not just about mathematics. It is where we decide which parts of the story are visible in the spreadsheet and which parts still need human follow-up.

Worked example: stage one preprocessing strip for several clusterers.

Step 1: Launch Orange and drag the `CSV File Import` or `File` widget onto the canvas. Point it at the engineered table supplied with the course materials.

Step 2: Connect `Select Columns` so latitude, adjusted longitude, turnover, and any other numeric drivers can be staged deliberately while identifiers stay out of the distance calculation.

Step 3: Add a `Preprocess` or `Normalize` step so scaling and imputation decisions live in one visible place instead of being scattered across widgets.

Step 4: Branch that same prepared data into `k-Means`, `DBSCAN`, and `Distances` feeding `Hierarchical Clustering`. This keeps the comparison fair because the clusterer changes while the preparation stays fixed.

Step 5: Keep a `Geo Map` or `Scatter Plot` downstream when you want to sanity-check the clusters against the city layout.

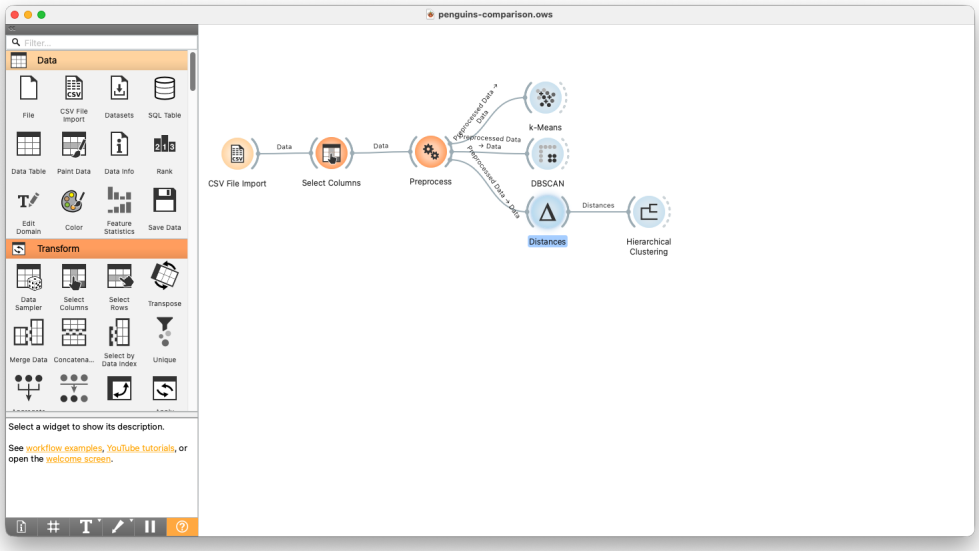


Figure 5.1: An Orange workflow that keeps one preparation strip and then swaps the clustering widget. The same staged data can feed `k-Means`, `DBSCAN`, or hierarchical clustering, which makes the comparison much easier to explain.

This sequence leaves room for further experimentation with optional widgets such as `Heat Map`, `Silhouette Plot`, or `Geo Map`. More importantly, it makes the teaching move explicit: once the preparation steps are visible, it becomes much easier to translate the same logic from Orange into Python without making the whole topic feel like a completely new subject.

Scaling changes the geometry. Once we move beyond plain latitude and longitude, the raw units stop being comparable. Turnover might be measured in dollars, distance in kilometres, and demographic variables in percentages. Centering by itself just recentres the ruler; it does not change pairwise distances. The real geometric shift comes when we scale the spread as well, so one wide-ranging feature does not dominate every distance calculation.

One point is easy to miss the first time we see centred data: zero has changed meaning. After centering, zero does not mean “nothing”; it means “typical for this dataset”. In

the classroom penguins example, the point $(0, 0)$ is the average penguin on both plotted measurements at once. In a retail table, the same logic says that a centred turnover value below zero is below the dataset average, while a positive value is above it.

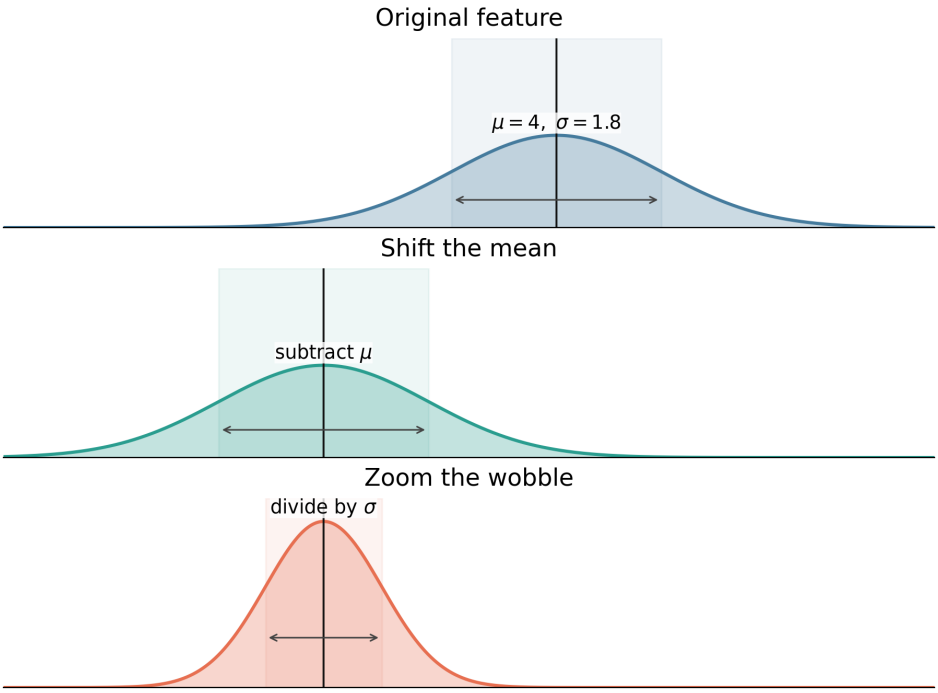


Figure 5.2: Shifting the mean recentres the axis; scaling the spread changes the geometry that a distance-based clusterer sees.

If the depot story depends only on local geography, leaving extra normalisation off can be sensible because the map scale is the story. The moment we mix geography with turnover, accessibility, or demographic indicators, we need to think harder. Standard scaling is the first baseline to try. If a few giant outliers are stretching the axis, robust scaling can be safer because it uses the median and the interquartile range. If a later step genuinely needs a fixed range, MinMax scaling is the more honest tool.

Missing values do not quietly disappear. Clustering still needs a complete numeric table. If turnover, accessibility, or a demographic field is missing, the distance calculation breaks. A median fill is usually the safest baseline when the numeric field is skewed. Mean imputation can be dragged around by a few large values, and KNN imputation is more local but can also blur the boundary between neighbouring clusters by inventing in-between values that were never really observed.

In other words: impute because the algorithm needs a number, but record the choice because the choice changes the geometry. Later, when this workflow turns into a train/test pipeline in Python, the imputer must be fitted on the training slice only. For the exploratory Orange workflow in this chapter, the important habit is to make the choice explicit and keep it visible on the canvas.

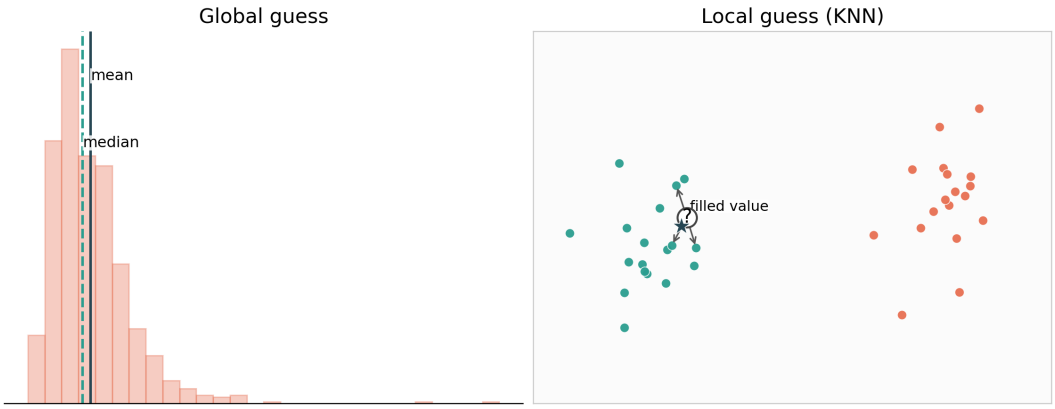


Figure 5.3: Imputation is not housekeeping; it changes the geometry before clustering even begins. Global fills and local fills solve different problems and create different risks.

5.3 Clustering Workflow and Decision Support

We operationalise the engineered features through k-means and translate the outputs into expansion recommendations.

Learning objectives

After reading this section, we will be able to:

- configure a k-means workflow in Orange that respects geographic scale and business constraints;
- evaluate candidate clusterings using silhouette diagnostics and scenario analysis;
- compare k-means with DBSCAN, hierarchical clustering, and mean shift when the business question changes;
- convert cluster assignments into depot placement guidance and follow-up investigations.

Prerequisites

Before starting, confirm that we can:

- explain how the k-means algorithm partitions observations around centroids;
- interpret silhouette scores as measures of cohesion and separation;
- build simple Orange workflows that chain File, k-Means, and visualisation widgets.

The workflow kicks off with a File widget pointing at the engineered dataset and a Select Columns widget that keeps the projected or locally adjusted spatial features while dropping identifiers from the distance math. Once the coordinates have been put on a

common linear scale, we sometimes avoid extra normalisation if geography alone is the story we want to preserve. When the feature set mixes geography with turnover or demographic fields, the preprocessing strip becomes part of the argument, not just a technicality. From there we experiment with the number of clusters, starting with two depots as a bare-bones network and nudging the count upward while watching the silhouette score.

Silhouette scores range from -1 to 1 and act like a comfort meter: values near 1 mean a store sits happily inside its cluster, numbers around 0 signal a border case, and negative values warn that the store probably belongs elsewhere.

When we switch from k-means to DBSCAN, the tuning story changes. Instead of choosing k, we choose a neighbourhood radius ε and a `min_samples` threshold. A practical classroom start is `min_samples` $\approx 2D$, where D is the number of numeric features. We then use a k-distance elbow plot as a guide for ε, but only as a guide. The real check is still the same mix of metrics and judgement: number of clusters, noise rate, silhouette on the non-noise points, and whether the map still tells a believable business story.

Illustrative scenarios

Example 1 – Northern corridor depot debate.

Step 1: Insert `k-Means`, keep only the projected or locally adjusted spatial features in the feature set, and disable extra normalisation so the depot spacing remains geographically meaningful.

Step 2: Branch the centroid output to a second `Geo Map` so we can contrast store clusters with suggested depot locations on side-by-side maps.

Step 3: Facilitate a quick reflection on whether a lone southern store justifies its own depot or whether we should revisit the feature mix before committing capital.

Example 2 – Silhouette-guided network sizing.

Step 1: Switch the `k-Means` widget to suggest cluster counts, surface the silhouette scores, and narrate the cost trade-off that keeps two depots attractive even when more clusters look tempting on paper.

Step 2: Keep the `Geo Map` visible while adjusting k live so we can compare how depot coverage shifts between two, three, and seven territories.

Step 3: Add a `Silhouette Plot`, link it back to the clustering, and highlight skinny bars to show which stores sit uneasily between territories before jumping back to the map for geographic context.

Step 4: Close by demonstrating how a stray categorical variable such as postcode can distort the centroids, then filter it out so the silhouette plot settles on meaningful territories again.

Framing silhouette analysis alongside the live map prepares readers to justify their chosen k clearly: diagnose ambiguous stores, revisit the feature set, and only then lock in a depot plan.

Silhouette diagnostics give us both numbers and pictures. Orange shows the average silhouette width for each run so we can see how cleanly separated the clusters are. We scan the plot for skinny bars—usually a hint that one store is stranded on its own. When that happens, we revisit the feature set to check for data issues or missing context. We also hook the cluster output into a Geo Map widget to colour regions by assignment and check whether the suggested centroids land in sensible spots along transport corridors rather than on top of existing stores.

Worked example: tuning k-means with diagnostics.

Step 1: Open your saved workflow and insert a `k-Means` widget after the feature engineering steps. Set the number of clusters to two to establish a baseline.

Step 2: Enable the "Silhouette" output on the `k-Means` widget and connect it to the `Silhouette Plot`. Note the average score and identify any thin bars indicating unstable assignments.

Step 3: Duplicate the `k-Means` widget, change k to three, and point it at the same feature set. Compare silhouette scores and check the `Geo Map` to see whether depot territories now respect natural transport corridors.

Step 4: Add a `Group By` (Aggregate) widget that groups on the cluster column and computes the mean delivery distance per cluster, then connect a `Data Table` to view the aggregated result. Use this table to craft a short recommendation memo for the logistics manager.

Step 5: Capture screenshots of both the silhouette output and the tuned parameter panel. They provide evidence for the final depot recommendation.

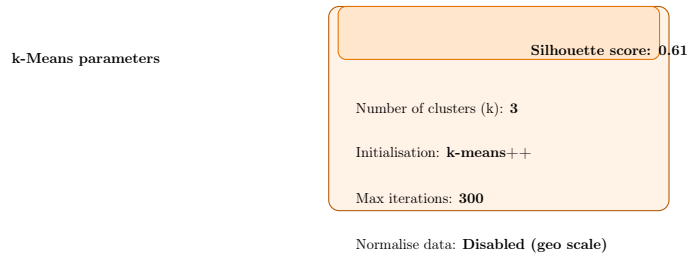


Figure 5.4: Annotated k-means configuration reflecting one tuned setup for the depot-planning exercise.

Troubleshooting sidebar.

- **"No data on input" warning:** Check that each widget has an active connection and that the `File` widget is pointed at the refreshed CSV, not a temporary preview file.
- **Silhouette plot shows negative bars:** Inspect the feature scaling. Revisit the `Formula` outputs or add a `Normalize` widget before k-means.
- **Geo Map missing tiles:** Switch the map provider to OpenStreetMap or refresh the internet connection proxy settings under Orange preferences.
- **Workflow crashes on save:** Remove absolute file paths in the `File` widget and use relative paths instead.

Beyond k-means: choosing a clustering method

K-means is the right anchor for this chapter because it is fast, easy to explain, and useful when we want compact territories around centroids. It is not the only clustering method we should know, though. In practice, the method depends on what kind of structure we think the data contains and what kind of decision we need to support.

K-means. Use k-means when you want every store assigned to a cluster and you are happy to choose the number of clusters k in advance. It works best when the clusters are fairly compact and the notion of a centroid makes business sense, such as placing depots for groups of nearby stores.

DBSCAN. Use DBSCAN when density matters more than centroids. It can discover irregularly shaped groups and mark isolated stores as noise instead of forcing every observation into a territory. That is useful when some stores are genuine outliers, but the method becomes sensitive to the distance scale and the `eps` or neighbourhood settings.

Hierarchical clustering. Use hierarchical clustering when you want a nested view of the data rather than one fixed answer. The dendrogram lets you cut the tree at different heights and ask whether the business prefers a coarse regional split or a finer local one. That makes it good for exploratory work and stakeholder discussion, even though you still need to justify where you cut the tree.

Mean shift. Use mean shift when you want clusters to emerge around dense peaks without choosing k first. The idea is appealing because it looks for high-density regions directly, but it depends heavily on the bandwidth setting and can become slow on larger datasets. In teaching, it is often most helpful as a contrast with k-means and DBSCAN: another reminder that clustering is about assumptions, not just button clicks.

> **Method-choice shortcut.** If the business wants compact territories with every store assigned, start with k-means. If it wants to separate dense regions

from noise, try DBSCAN. If it wants a nested picture that can be cut at several levels, use hierarchical clustering. If it wants clusters to form around density peaks without fixing k, mean shift is the natural comparison point.

What the hands-on practical adds

The hands-on practical makes the same ideas physical before we trust the software. One small tabletop dataset gets reused three ways: DBSCAN asks which numbered tags sit within one shared radius, hierarchical clustering records the merge order, and mean shift nudges a probe toward a local density peak. The point of the practical is not mathematical precision. The point is to make each algorithm's assumption visible.

DBSCAN by hand. One paddle-pop stick gives the whole group a shared ε distance. Tags with enough neighbours inside that stick-length become core points; nearby stragglers become border points; genuinely isolated tags stay noise. That makes the density rule visible without hiding it inside a widget panel.

Figure 5.5: A tabletop DBSCAN analogue: one paddle-pop stick supplies the shared radius, so the core-border-noise distinction can be checked by eye.

Hierarchical clustering by hand. The same numbered beekeeping tags also make a good single-linkage exercise. Students find the closest pair, record it in a merge log, then sketch the dendrogram from that log. The useful insight is that hierarchical clustering gives a tree that can be cut at several heights, not one magically correct answer.

Mean shift by hand. Mean shift becomes less mysterious when we turn it into repeated local averaging. Place a circular neighbourhood over the tags, estimate the visual centre of the points inside it, move the probe, and repeat until it barely shifts. Starting from a second place then shows whether two probes settle on the same density peak or on different ones.

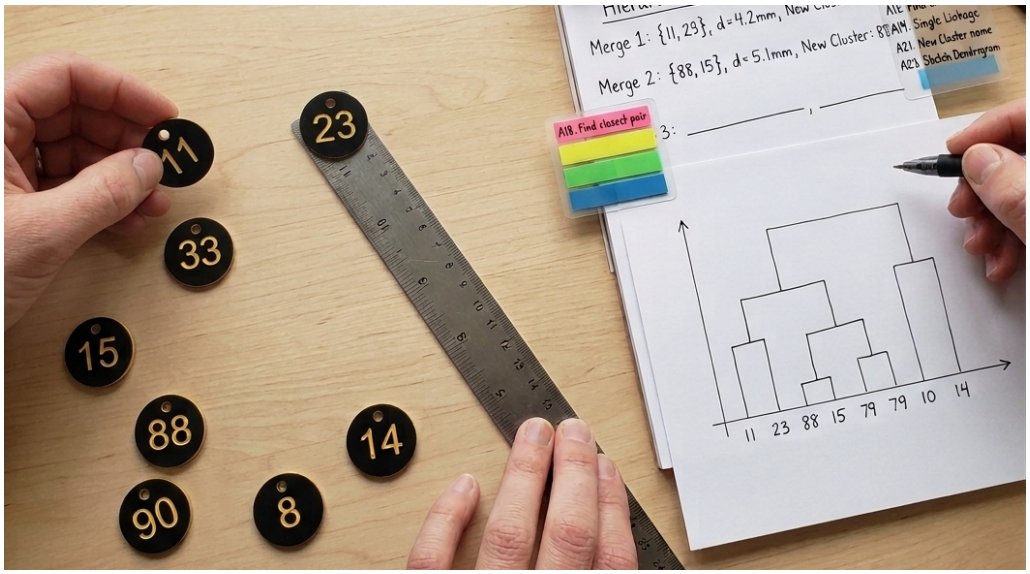


Figure 5.6: The hierarchical-clustering practical keeps a merge log beside the tabletop layout so the dendrogram grows from recorded distance decisions rather than from black-box software output.

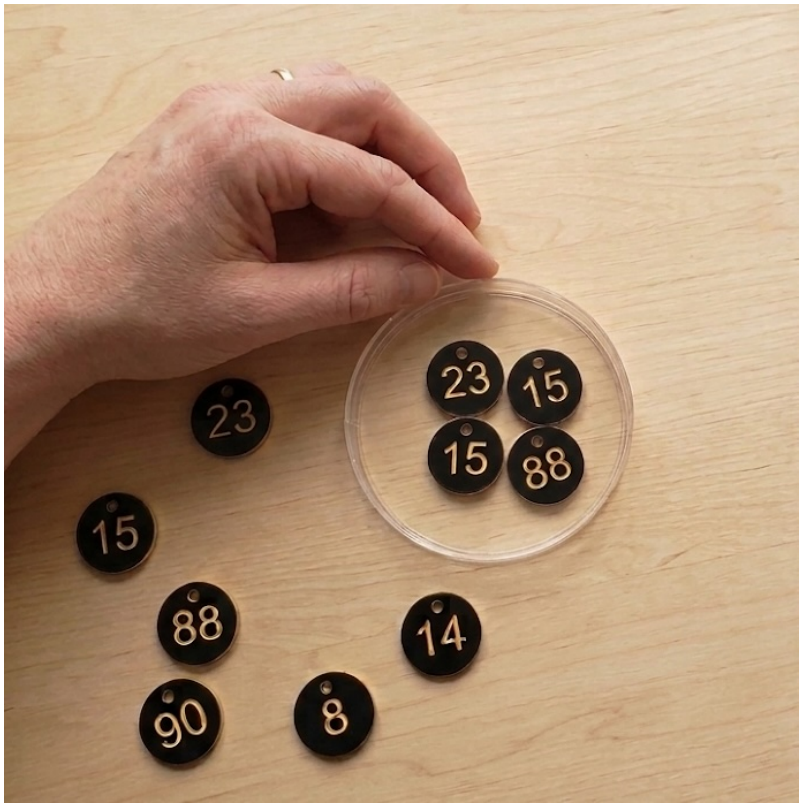


Figure 5.7: A simple lid or circular marker turns mean shift into a visible "move to the local centre" process rather than an abstract density-peaks slogan.

Why the tuning story matters

K-means is doing two jobs at once here. On the technical side, it is trying to keep each store reasonably close to the centroid of its assigned territory. On the business side, we are asking whether those territories are believable enough to support a depot decision. That is why the silhouette score matters, but it is also why the score is never the whole story. A mathematically neat split can still be a bad logistics plan if it ignores roads, warehouse constraints, or local knowledge.

Section summary

- Scaling and imputation choices change the geometry before any clusterer runs, so they belong in the argument, not just in the setup notes.
- Configuring k-means in Orange requires careful feature selection, scale management, and iterative tuning.
- Other clustering methods encode different assumptions: DBSCAN emphasises density and noise, hierarchical clustering emphasises nested structure, and mean shift emphasises density peaks.
- Silhouette plots and geo-visualisations expose when a clustering produces unstable or impractical territories.
- Scenario analysis connects the technical output to depot planning decisions and follow-up fieldwork.

Self-check questions

1. Which business constraint would prompt you to accept a lower silhouette score in exchange for more clusters? *Hint: revisit service-level targets and delivery-time thresholds.*
2. How will you validate the recommended depot locations with stakeholders before committing capital expenditure? *Hint: outline a pilot or field validation plan.*
3. What does a negative silhouette bar tell you about a store's assignment, and how would you respond? *Hint: consider both feature engineering and cluster count adjustments.*
4. Why can a mean fill or an unscaled high-range feature change the cluster assignment before you switch algorithms? *Hint: think about how the distance calculation is built.*
5. Which clustering method would you try first if several stores looked like genuine outliers rather than members of a compact territory, and why? *Hint: think about how the method treats noise points.*

6. Describe one optimisation insight from the "Why the tuning story matters" subsection in your own words. *Hint: link the k-means objective to a business-friendly explanation.*

7. Propose a short troubleshooting checklist to run before presenting results to leadership. *Hint: include both Orange diagnostics and data refresh checks.*

Self-check reflections

1. Service guarantees like maximum delivery times or per-depot load caps might justify extra clusters even if the silhouette drops.

2. Plan a depot pilot, gather manager feedback, and compare delivery metrics before and after to confirm the recommendation lands well.

3. A negative bar signals a store fits another cluster better; revisit the features or try a different k to see whether the store finds a clearer home.

4. Both the fill value and the scaling step change the coordinates that the algorithm actually sees, so they can move border cases from one cluster to another before the clusterer itself changes.

5. DBSCAN is the best first candidate because it can leave genuine outliers unassigned as noise instead of forcing them into a misleading compact cluster.

6. Explaining that k-means balances keeping stores close to a centroid against splitting them too finely reframes the optimisation math as a trade-off between efficiency and complexity.

7. Check widget connections, confirm the latest CSV is loaded, review silhouette plots for outliers, and document any manual adjustments before presenting.

Chapter exercises

1. Turn the supermarket-location story into a decision question, a unit of analysis, and two risks of using geography alone.

2. Prepare a feature-role table for the supermarket data. Mark each field as feature, metadata, target-like context, or excluded, and justify two of the harder choices.

3. Plot the store locations and inspect them for geocoding or scale problems. Record one check you trust and one check that still needs local knowledge.

4. Compare k-means with one alternative clustering method on the same prepared data. Explain whether the alternative changes the business recommendation or merely changes the visual grouping.

5. Write a short depot or territory memo that includes the chosen cluster count, the recommended next step, and one uncertainty that should be checked before acting.

Chapter 6

Exemplar-Based Clustering with Isolation Kernels

Chapter overview

The clustering chapters so far have mostly relied on numeric geometry: rows become points, distances compare points, and algorithms group nearby points. Exemplar-based clustering starts from a different question: if we show each observation a small set of reference examples, which reference example is it most similar to? Repeating that question with many small reference sets turns a messy object into a categorical signature. This chapter develops that classroom intuition, connects it to isolation kernels and point-set kernel clustering, and outlines practical activities for mixed data that may include missing values, categories, short text, or images.

6.1 Turn Similarity Questions into Categorical Codes

Represent messy observations by repeatedly asking which exemplar they most resemble.

Learning objectives

After reading this section, you will be able to:

- describe how repeated exemplar panels convert observations into categorical signatures;
- explain why this representation can be useful when ordinary numeric distances are awkward;
- calculate a simple matching score between two exemplar signatures.

Prerequisites

Before starting, make sure you can:

- explain why scaling and missing values affect distance-based clustering;
- compare categorical values with a match or mismatch score;
- interpret a small clustering output from chapter 5.

Many real datasets resist the tidy "one row, all numeric columns" story. A record might contain a suburb, a price band, two missing values, a product photograph, and a short customer note. We can force that into numbers, but the resulting distance may be more a record of our preprocessing choices than of meaningful similarity.

An exemplar panel offers a different route. Choose a small set of reference items, such as A, B, C, and D. For each observation, ask, "Which of these reference items is it most similar to?" The answer is a category: A, B, C, D, or perhaps "none". Then draw a new panel and ask again. After several panels, each observation has a signature such as `B-A-D-C`. Two observations are similar if their signatures match often.

Item	Panel 1	Panel 2	Panel 3	Panel 4	Signature
Ticket 07	B	A	D	D	B-A-D-D
Ticket 19	B	C	D	D	B-C-D-D
Ticket 24	A	C	B	A	A-C-B-A
Ticket 31	B	A	D	C	B-A-D-C

Table 6.1: A toy exemplar-code table. Tickets 07 and 31 match on three of four panels, while ticket 24 follows a very different pattern.

The matching score in Table 6.1 is just the proportion of panel answers that agree. Ticket 07 and Ticket 31 agree on panels 1, 2, and 3, so their similarity is $3/4$. Ticket 07 and Ticket 24 agree on none of the four panels, so their similarity is $0/4$. This calculation is simple enough to do by hand, but it hints at a bigger idea: similarity can be built from repeated local comparisons rather than one global numeric formula.

Why the idea helps. Each panel asks for a relative judgement inside a small context. A person can compare three product descriptions more easily than define a universal product-distance formula. A language model can compare short records with mixed text and categories more naturally than it can justify a Euclidean distance across hand-encoded features. A domain expert can say that one case is most like exemplar B because both involve urgent cancellation, even if one record has missing price data and the other has a photograph.

The categorical signature is not magic. It moves the hard work into the similarity judgement and panel design. But that move is often useful. We can audit the panels, repeat the process with different random seeds, and inspect which exemplar choices drive a cluster.

Section summary

- Repeated exemplar panels turn each observation into a categorical signature.
- Signature matching gives a simple similarity score that works before we choose a numeric embedding.
- The method is attractive for mixed or incomplete data, but the exemplar-choosing process must be stable and auditable.

Self-check questions

1. Why might an exemplar panel be easier to explain than a hand-built mixed-data distance formula?
2. How would the similarity between two signatures change if we allowed a "none of these" answer?

6.2 Connect the Classroom Idea to Isolation Kernels

See isolation kernels as repeated random partitions that turn point similarity into shared-cell evidence.

Learning objectives

After this section, you will be able to:

- describe an isolation kernel without relying on advanced kernel-method vocabulary;
- explain how random exemplar subsets create a sparse feature representation;
- identify the difference between the classroom exemplar code and the formal kernel construction.

Prerequisites

Before proceeding, confirm that you can:

- explain how nearest-neighbour methods compare one point with another;
- describe k-means centroids and DBSCAN neighbourhoods at a high level;
- read a simple indicator-function formula.

Isolation kernels formalise a related idea. We repeatedly draw a small random subset of data points as reference points. That subset partitions the space into regions, often by assigning every point to the reference point that isolates or owns it under the chosen partition rule. Two observations are considered similar when they often fall into the same region across many random partitions.

A simple version can be written as:

$$K(x, y) = \frac{1}{t} \sum_{r=1}^{t} \mathbf{1}\{h_r(x) = h_r(y)\},$$

where $h_r(x)$ is the region label assigned to point x by random partition r, t is the number of partitions, and $\mathbf{1}\{\cdot\}$ is 1 when the condition is true and 0 otherwise. If two points repeatedly land in the same isolation region, the kernel score is high. If they rarely do, the score is low.

The important teaching point is that the kernel is *data dependent*. A Gaussian kernel uses the same distance rule everywhere once its bandwidth is set. An isolation kernel changes with the sampled reference points and the distribution of the dataset. Sparse regions and dense regions are not treated as though one global ruler tells the whole story. Research on isolation kernels reports that this data-dependent behaviour can improve large-scale online kernel learning and hierarchical clustering in settings where ordinary distance measures struggle [14, 16].

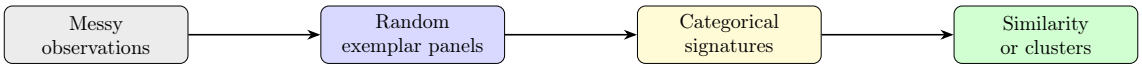


Figure 6.1: Exemplar panels create categorical signatures, which can then be compared or clustered. Formal isolation kernels give one principled version of this pattern.

The classroom version is intentionally looser than the formal kernel. In class, the "which exemplar is most similar?" judgement might be made by learners, a rubric, or a simple mixed-feature scoring rule. In the formal method, the partition and feature map are mathematically specified. That difference matters. The classroom version is a bridge for intuition and practical design; the formal kernel is the object we would cite and implement in a research-grade algorithm.

Section summary

- Isolation kernels compare points by asking how often random partitions place them in the same region.
- The resulting feature representation is sparse, finite, and shaped by the data distribution.
- A classroom exemplar-panel exercise is an analogy and practical scaffold, not a full replacement for the formal method.

Self-check questions

1. What does it mean for two points to share an isolation region?
2. Why might a data-dependent kernel behave better than one fixed global distance rule?

6.3 Use Point-Set Thinking to Grow Clusters

Compare a new object with a cluster represented by several exemplars rather than with one centroid.

Learning objectives

After this section, you will be able to:

- explain the point-set similarity question in plain language;
- contrast point-set clustering with centroid-based clustering;
- describe why point-set thinking suits clusters with several local prototypes.

Prerequisites

Before starting, ensure you can:

- explain why a centroid may be a poor representative for an irregular cluster;
- describe hierarchical clustering as a process of building groups from smaller groups;
- interpret a cluster as a set of examples rather than only as a label.

Point-set kernel clustering changes the question from "How similar are these two objects?" to "How similar is this object to that set of objects?" [15]. That is a small language shift with a useful modelling consequence. A cluster does not need to be compressed into one centroid. It can be represented by the examples it has already recruited.

This helps when a group has several faces. Customer complaints about delivery problems may include late parcels, damaged parcels, and missed notifications. A centroid-like summary can blur those stories together. A point-set view says that a new complaint belongs near the cluster if it resembles the set, even if it is closest to only one local part of the set.

A teaching version. Start with one seed item for each candidate cluster. For every unassigned item, compare it with the current set of examples in each cluster. Recruit the item into the cluster whose set it most resembles, then update the set. Repeat until the clusters stop changing or no confident assignment remains. This is not the full published algorithm, but it expresses the central idea: a cluster can grow by asking whether a point resembles a collection, not by measuring distance to a single average.

Connection to earlier clustering methods. K-means asks each point to choose the nearest centroid. DBSCAN asks whether a point sits in a dense neighbourhood. Hierarchical clustering asks which points or groups should merge next. Point-set kernel clustering asks whether a point is similar to the set that already characterises a cluster. Each method encodes a different idea of what a cluster is.

For mixed data, the point-set question is especially helpful. A cluster can retain several exemplars that cover different formats: one mostly numeric row, one text-heavy row, one image-rich row, and one row with missing fields. The similarity judge can compare the new item with whichever exemplars are informative rather than forcing every record through the same complete feature vector.

Section summary

- Point-set clustering compares an object with a group of objects, not just with another object or a centroid.
- Cluster sets can represent multi-prototype groups more naturally than a single average.
- The classroom version should be labelled as an intuition-building procedure unless it implements the formal point-set kernel.

Self-check questions

1. When would a cluster set be more informative than a centroid?
2. How does the point-set question differ from ordinary nearest-neighbour comparison?

6.4 Classroom Practical: Exemplar Ballots

Build clusters by voting on similarity to repeated exemplar sets.

Learning objectives

After this section, you will be able to:

- run a hands-on exemplar-ballot practical using cards or a spreadsheet;
- convert repeated exemplar choices into a code table;
- compare hand-built clusters with algorithmic or stakeholder groupings.

Prerequisites

Before running the practical, we should be able to:

- record categorical data consistently in a table;
- calculate simple proportions;
- discuss whether two messy records feel similar and justify the judgement.

The practical works best with objects that have several kinds of evidence. A set of fictional support tickets might include a short message, a product category, a rough account age, a missing value flag, and a tiny icon representing an attachment. A set of museum-object cards might include an image, material, century, region, and catalogue note. A set of product cards might include a photograph, a price band, review snippets, and tags. The data should be safe, fictional or public, and small enough to handle on a table.

Round structure.

Step 1: Deal each group the full object deck and a recording sheet.

Step 2: Draw three to five exemplar cards and label them A, B, C, and so on.

Step 3: For every other card, choose the exemplar it most resembles. Record only the exemplar label, not a long explanation.

Step 4: Return the exemplar cards to the deck, reshuffle, and draw a new panel.

Step 5: Repeat for six to ten panels, producing a signature for every object.

Step 6: Cluster cards whose signatures match often. Borderline cards should be flagged rather than forced.

Step 7: Name each cluster by inspecting its members and writing a short evidence note.

The activity is quick because each judgement is local. We do not need to design a universal distance formula across text, images, categories, and missing values. We only need to answer a repeated local question: "Of these exemplars, which one is most similar to this item?" The repeated answers accumulate into a structured representation that can be counted, compared, and challenged.

Spreadsheet extension. If we enter the signatures into a spreadsheet, we can compute pairwise matching proportions. For two rows, count how many panel columns match and divide by the number of panels where both rows have an answer. Sorting by that score gives nearest neighbours under the exemplar code. A simple hierarchical clustering can then use $1 -$ matching proportion as a distance, provided the limitations are clearly stated.

Debrief questions.

- Which clusters felt stable across panels, and which depended on a few lucky exemplar choices?
- Did the "none of these" option help, or did it become a way to avoid hard decisions?
- Which objects were repeatedly borderline, and what extra evidence would help classify them?
- How would the clusters change if the similarity judge cared more about text than images, or more about category than price?

This is also a useful place to discuss algorithmic responsibility. Exemplar choices are not neutral. If the exemplar panels overrepresent one customer type, one region, or one visual style, the signatures will encode that bias. Repeating panels reduces dependence on any single draw, but it does not remove the need to inspect the exemplar pool.

6.5 Use Cases and Guard Rails

Apply exemplar-code clustering when mixed evidence matters, but validate stability before trusting the clusters.

Learning objectives

After this section, you will be able to:

- identify suitable use cases for exemplar-based signatures;
- list the main risks in exemplar-panel design;
- design stability checks for repeated-panel clustering.

Prerequisites

Before starting, make sure you can:

- explain why unsupervised clusters need external interpretation;
- compare two clusterings at a high level;
- document preprocessing choices for a modelling workflow.

Exemplar-code clustering is worth considering when ordinary preprocessing is doing too much damage. Mixed support tickets, safety incident notes, product catalogues, student feedback themes, image-and-text records, or partially missing operational logs all fit the pattern. We want structure, but we may not want to pretend that a single numeric feature space is already obvious.

The guard rails are equally important. First, define who or what makes the similarity judgement. A rubric, a trained human group, a deterministic embedding model, or a language model will produce different signatures. Second, repeat the process with several random seeds or panel draws. Stable clusters should keep reappearing. Third, inspect cluster names with domain experts. Unsupervised labels are proposed categories, not ground truth. Fourth, preserve examples from each cluster so the grouping remains explainable.

> **Practical rule.** Use exemplar signatures as a way to open up messy clustering problems, not as a way to avoid validation. If the clusters will influence services, pricing, safety, or access, they need the same review and governance as any other model output.

Chapter summary

- Repeated exemplar panels can turn mixed observations into categorical signatures.
- Isolation kernels provide a formal version of repeated random partitions and shared-region similarity.
- Point-set kernel clustering compares a point with a set, allowing clusters to grow around several exemplars rather than one centroid.
- The exemplar-ballot practical gives us a hands-on way to see how local similarity judgements become cluster structure.

Key term glossary

Exemplar panel A small reference set used to ask which example a new item most resembles.

Categorical signature The sequence of exemplar labels assigned to an observation across repeated panels.

Isolation kernel A data-dependent kernel that compares points by how often random partitions place them in the same region.

Point-set kernel A similarity measure that compares one object with a set of objects.

Panel bias Distortion caused by an exemplar pool or draw process that overrepresents some kinds of items.

Extension challenges

- Build a small spreadsheet that computes pairwise matching scores from exemplar signatures.
- Compare exemplar-code clusters with k-means, DBSCAN, or hierarchical clustering on a mixed dataset after one-hot encoding.
- Run the exemplar-ballot practical twice with different panels and measure which clusters remain stable.
- Replace human similarity judgements with a documented rubric or model, then compare the resulting signatures.

Conclusion and next steps

Exemplar-based clustering is a useful bridge between intuitive human comparison and formal kernel methods. It keeps the practical question visible: "What is this most similar to, among these examples?" Repeating that question turns messy evidence into something we can count, cluster, and debate. The next chapters continue the same discipline with regression and classification: choose a representation that matches the decision problem, then validate the consequences rather than trusting the representation by default.

Chapter exercises

1. Given eight short evidence cards, create three categorical similarity codes that could be used in an exemplar ballot.
2. Choose two exemplars from the card set. Explain why each exemplar represents a different kind of cluster rather than merely an extreme case.
3. Work through a tiny isolation example: if two records are separated early by several random partitions, what does that suggest about their similarity?
4. Start with one exemplar and decide which records should join its point-set cluster. Mark one borderline record and explain what extra evidence would settle it.
5. Name one setting where exemplar clustering would be easier to explain than k-means and one setting where it would be a poor fit.

Chapter 7

Hands-on Activity 2: Linear Regression by Hand

This activity uses everyday objects to make regression feel less abstract. We begin with a plain baseline line, then explore two robust approximations so we can see why outliers matter before the formal regression chapter turns the same ideas into software.

7.1 Kit Checklist

Each group needs:

- numbered beekeeping tags or similar tokens to stand in for data points; the handy range is 11–66 plus 80–89 for the hold-out set;
- two dice for quick random sampling;
- thread, tape or reusable putty, and scissors;
- a ruler or straight-edge;
- a comb or narrow card for the RANSAC-style inlier count;
- paper or a whiteboard for recording trial results.

7.2 Welcome and Setup

1. Form groups of around five learners and check that each group has the full kit.
2. Arrange the numbered tokens in a rough upward trend with a few clear outliers.
3. Reserve tokens 80–89 as a tiny test set. The rest are your training points.
4. Explain the order of the activity: ordinary line first, then Theil–Sen, then RANSAC.

7.3 Stage 1: Fit a Baseline Line First

Start with the simplest possible question: what line would a reasonable person draw through the main pattern?

1. Treat the edge of the table as the x-axis and the perpendicular direction as the y-axis.
2. Stretch a piece of thread across the middle of the main cloud of training points.
3. Adjust it until the line seems to balance the points above and below it as fairly as you can.
4. Measure the vertical distance from each reserved test token (the tokens 80–89) to the line.
5. Square those distances, add them, and divide by ten to estimate the mean squared error.
6. Write down one sentence describing what felt hard about drawing the line by eye.

The point of this step is to make the baseline visible.

7.4 Stage 2: Approximate Theil–Sen

Now switch to a median-based idea.

1. Roll the dice twice to choose two training tokens.
2. Stretch the thread through those two points and extend the line across the table.
3. Repeat this seven to eleven times and keep each candidate line, even if some are obviously poor.
4. Compare the candidate lines and choose the "middle" one: the line that best represents the median trend among the samples.
5. Measure the mean squared error on the reserved test tokens (80–89) and compare it with the baseline line.
6. Photograph the final layout.

Teaching simplification: the full Theil–Sen estimator considers many pairwise slopes systematically. This tabletop version is a classroom analogue that captures the core idea without reproducing the exact algorithm.

7.5 Stage 3: Approximate RANSAC

RANSAC focuses on agreement. Instead of looking for a middle slope, it keeps asking which candidate line attracts the strongest inlier set.

1. Thread the line through the middle teeth of the comb or use a narrow straight-edge as an inlier guide.

2. Choose a random pair of training tokens and rebuild the candidate line through them.

3. Slide the comb along the line and count how many training points lie close enough to count as inliers.

4. Record the inlier count, then repeat with fresh random pairs.

5. After around ten trials, keep the line with the strongest consensus.

6. Estimate the mean squared error on the reserved test tokens and compare it with the baseline and Theil–Sen lines.

Teaching simplification: real RANSAC uses an explicit inlier threshold and usually refits the final model on the inlier set. This activity keeps only the consensus idea.

7.6 Stage 4: Recreate the Baseline in Orange

Orange's role in this activity is to make the ordinary regression workflow visible. The robust comparison itself belongs in the later Python extension.

1. Enter the rough (x, y) layout using `Paint Data` or another simple data-entry route.

2. Connect the data to `Scatter Plot` so everyone can compare the digital view with the table layout.

3. Use `Select Columns` to set x as a feature and y as the target.

4. Fit `Linear Regression` and inspect the output in `Predictions` and `Data Table`.

5. Compare the ordinary software line with the baseline line you drew by hand.

6. Discuss which points seem to be pulling the software line hardest.

Figure 7.1: Hands-on regression kit with numbered tags, thread, Blu Tack, and comb laid out on a table, ready for the baseline, Theil–Sen, and RANSAC approximations.

Chapter exercises

1. Draw an ordinary baseline line through the activity points. Record two reasons the line seems plausible or weak.

2. Choose at least five point pairs, compute their slopes, and take the median slope as a Theil–Sen-style approximation.

3. Try one RANSAC-style candidate line. Mark which points are inliers under a fixed tolerance and decide whether the line should survive.

4. Recreate the ordinary baseline in Orange. Compare the software line with the physical approximation and explain the largest difference.

Chapter 8

Linear Regression First, Then Robust Alternatives

Chapter overview

The earlier hands-on linear-regression chapter gave the physical version of this story. Here we formalise it. The core path introduces ordinary least squares (OLS) as a baseline regression model, then explains when that baseline is likely to mislead. Orange now has a course add-on for the robust estimators we need, so the same canvas can compare OLS with Huber, Theil–Sen, and RANSAC before the later computing-extension chapter (chapter 21) moves into Python. The main lesson is not "robust is better". The main lesson is "start with a baseline, diagnose the problem, and then choose the tougher method only if the data justify it".

8.1 Start with an ordinary straight line

Linear regression tries to draw one straight line through the data so that the predictions are as close as possible to the observed values. In a housing example, we might ask whether larger blocks tend to have higher prices, or whether distance from the CBD tends to reduce price per square metre. If the pattern is roughly straight, a simple line gives us two useful things at once:

- a prediction for new cases;
- a compact description of the overall trend.

In Orange, this baseline is easy to inspect. Load the staged dataset from chapter 3, use **Select Columns** to mark the target clearly, fit **Linear Regression**, and inspect the result in **Predictions**, **Data Table**, and a scatter plot. If the **Robust** add-on is installed, the same workflow can add **Robust Regression** and choose **Huber**, **Theil-Sen**, or **RANSAC**. At this stage we are not chasing sophistication. We are asking a simpler question: does a straight line look like a reasonable first summary of the relationship?

That question matters because linear regression is often good enough. If the data are fairly clean and the trend is broad rather than twitchy, the ordinary line is easier to explain than a more resistant alternative. A model that is slightly less clever but much easier to explain is often the better teaching example and sometimes the better business tool as well.

> **Optional maths note.** A residual is the vertical gap between an observed value and the value predicted by the fitted line. Ordinary least squares chooses the line that minimises the sum of squared residuals. Squaring the residuals makes large mistakes count much more than small ones, which is exactly why extreme points can dominate the fit.

8.2 Notice when OLS stops being trustworthy

The ordinary line becomes less trustworthy when a small number of observations are extreme enough to drag it away from the broader pattern. A luxury property, a redevelopment site, a recording mistake, or a parcel with highly unusual zoning can all have that effect.

There are a few warning signs worth learning early:

- The line seems to follow one or two unusual points more than the main cloud.
- Errors for ordinary cases become much larger after a handful of extreme observations are added.
- The spread of the residuals changes a lot across the range of the data.
- Different subgroups clearly follow different stories, so one straight line is trying to do too much.

8.3 Add robust methods only when they earn their place

If the main pattern is still roughly linear but a minority of points are clearly disruptive, robust regression becomes worth discussing.

Theil–Sen. Theil–Sen estimates the line from many pairwise slopes and then uses a median-based summary. The practical advantage is that a few extreme cases do much less damage than they would under OLS.

RANSAC. RANSAC repeatedly samples small subsets, fits a candidate line, and asks how many observations agree with it closely enough to count as inliers. It is especially useful when the data contain a clear main trend plus a visible set of outliers.

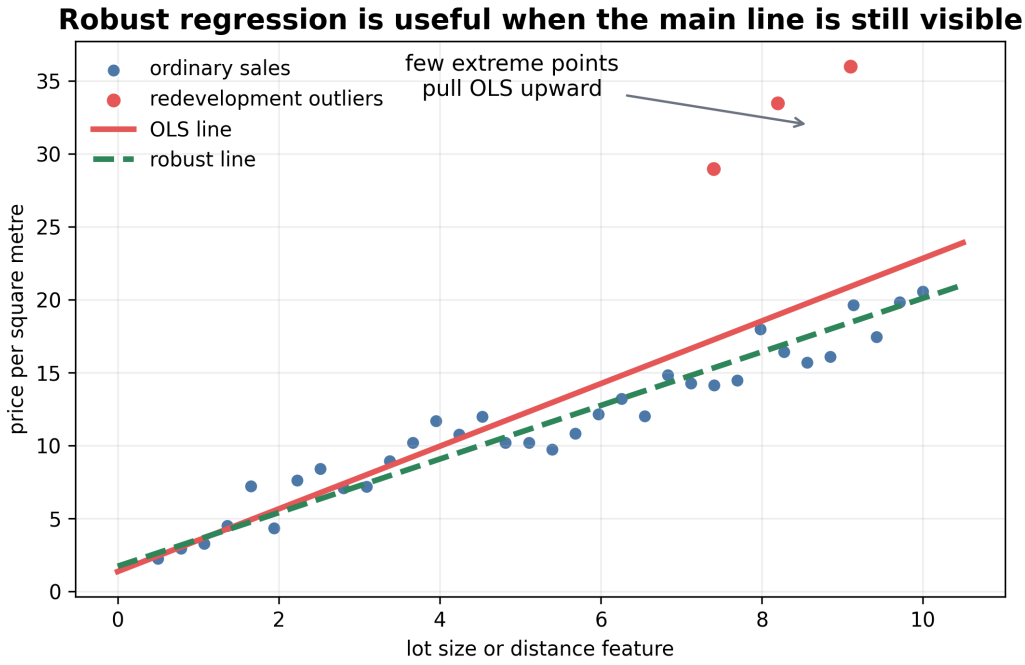


Figure 8.1: A few extreme observations can pull the ordinary least-squares line away from the main cloud. A robust comparison is useful only when the main linear pattern is still visible and the question calls for ordinary-case stability.

These are not default replacements for OLS. They are specialised tools for a narrower situation: the linear story is still mostly right, but a minority of points are making the ordinary line unstable.

> **Orange workflow note.** Install the course add-on from `https://github.com/solresol/Orange3-Robust`. The **Robust Regression** widget outputs both an Orange learner for **Test and Score** and annotated data containing predictions and residuals. Use its built-in scaling checkbox instead of placing a separate scaler directly in front of it.

If we want the code-based side-by-side comparison, we can jump ahead to chapter 21. In this main-path chapter we stay with the conceptual judgement: diagnose the ordinary line first, compare it with a robust alternative in Orange, then keep the tougher method only if the data earn that extra machinery.

8.4 Worked example: ordinary housing, unusual sales

Suppose we are modelling price per square metre for a small housing sample. Most homes follow a broad pattern, but a few redevelopment sites sell for unusually high amounts because buyers are paying for future potential rather than for the current dwelling. Those cases can yank an OLS line upward and make ordinary suburban homes look more expensive than they really are.

In a case like that, a robust estimator may give the more useful everyday forecast. That does not mean the redevelopment sites are "bad" data. It means they answer a different commercial question. A stakeholder pricing family homes may want a stable model for ordinary cases. A developer may want the opposite and care mainly about those high-value exceptions. The method only makes sense once the question is clear.

8.5 How to report the result

When we write up a regression comparison, we keep the story plain:

- say what the baseline was;
- say what problem we observed;
- say what changed after the robust comparison;
- say whether the difference actually matters for the decision at hand.

For the Orange-based version of the story, that report can stay at the level of diagnosis and model choice. The later Python extension shows how the same comparison looks when the estimators are run side by side in code.

Chapter exercises

1. Inspect a small residual plot or residual table. Decide whether ordinary least squares is trustworthy and name the evidence you used.
2. Remove or downweight one suspicious observation in a regression sketch. Describe how the fitted line should move and why.
3. For three short scenarios, choose ordinary least squares, Theil–Sen, RANSAC, or data cleaning before modelling. Justify each choice in one sentence.
4. Write a short reporting paragraph that compares the ordinary line with a robust alternative without claiming that robust methods are automatically superior.

Chapter 9

Hands-on Activity 3: Logistic Regression by Hand

We turn the language of logistic regression into a tactile game. We place chess pieces on a board and slide markers along an S-shaped curve. That hands-on setup lets us feel how probabilities change before the formal chapter repeats the same moves in software.

This activity gives a physical preview of log-odds, likelihood, thresholds, and class probabilities. The later Orange workflow then has something concrete to connect back to.

9.1 Kit Checklist

Stock each group with the following items so they can run the logistic-regression game on a physical chessboard before moving to Orange.

- **One chessboard or printed 8×8 grid.** The board provides the coordinate system for the activity.
- **A small set of contrasting pieces or tokens.** Use two colours so the positive and negative classes remain obvious.
- **A printed logistic-curve sheet.** Students use this as a shared probability reference; the textbook reproduces it in Appendix A.1 on p. 286.
- **Markers, pegs, or sticky notes.** These help groups track the intercept and any coefficient changes they try.
- **Paper or a whiteboard.** Each group needs somewhere to record coordinates, coefficients, and likelihood calculations.
- **Optional string or ruler.** This helps when students sketch the decision boundary back onto the board.

Appendix A.1 keeps a print-ready copy of the reusable logistic-curve poster inside the textbook itself for tutors and workshop prep.

9.2 Stage 1: Build the Chessboard Dataset

Gather each group around a table with the chess set or printed grid, the logistic-curve sheet, a few markers, and somewhere to write.

1. Treat the chessboard as an 8×8 grid of *squares*. Index the square centres with files $x = 0, \ldots, 7$ and ranks $y = 0, \ldots, 7$. Place five mixed-colour pieces on distinct squares; duplicates are fine as long as you track which is which.

2. Label white pieces as $y = 1$ (positive class) and black pieces as $y = 0$ (negative class). Encourage groups to include one point near $(0, 0)$ and another along the same file so the intercept and slope feel concrete.

3. Record each piece's coordinates on the whiteboard. When students move a piece, update the list so the dataset stays consistent.

4. Park the twin of every chess piece on the logistic curve. Keep a marker or sticky note handy to mark the intercept position on the curve.

9.3 Stage 2: Initialise a Logistic Model

Give each group a shared starting point so comparisons make sense.

- Initialise the intercept at 0 with both coefficients set to 0.1.

- Write the working formula $s = \beta_0 + \beta_{\text{rank}} \times \text{rank} + \beta_{\text{file}} \times \text{file}$ on the whiteboard. Remind everyone that s is in log-odds—a scale where we add numbers, then convert them to probabilities using the S-shaped logistic curve.

- Peg the intercept onto the curve and place each twin piece at the probability it implies.

Talk through the interpretation: a +1 step in log-odds multiplies the odds by e^1, whereas a $+\ln 2$ step doubles them. Log-odds simply means "how tilted are the chances in favour of the positive class"—positive numbers favour white pieces, negatives favour black ones.

9.4 Stage 3: Calculate Likelihood by Hand

Walk the groups through the loss calculation so they can judge improvements.

1. For each chess piece, substitute its coordinates into the linear predictor to compute s.

2. Read the corresponding probability p from the logistic curve. For white pieces keep p; for black pieces use $1 - p$.

3. Multiply the five contributions together to obtain the joint likelihood the model assigns to the observed colours.

4. Record the product on the whiteboard. Challenge groups to beat their previous best as they refine the model.

9.5 Stage 4: Greedy Coordinate Search

Let each group iterate on the coefficients to improve the likelihood.

1. Take the pieces off the logistic curve but leave them on the board.
2. Propose a small tweak to one parameter at a time.
3. Recompute s and the likelihood. If the product increases, keep the change; otherwise revert and try a different direction.
4. Repeat until the improvements taper off. Record the final parameters so everyone can compare results later.

This is a deliberately simplified hand-worked approximation to fitting logistic regression. Orange will optimise the coefficients more systematically once we move into software.

9.6 Stage 5: Evaluate the Classifier

Once the group settles on parameters, they can translate the model back into familiar metrics.

1. Use the model to label each chess piece as white (probability > 0.5) or black (probability ≤ 0.5). Tally true positives, false positives, true negatives, and false negatives on the whiteboard.
2. Compute accuracy, precision, and recall from the tallies. Discuss how moving the threshold off 0.5 would change the counts.
3. Stretch a piece of string across the chessboard where $\beta_0 + \beta_{\text{file}} \times x + \beta_{\text{rank}} \times y = 0$. This visual decision boundary helps explain how the model separates the two colours.

9.7 Stage 6: Explore Variations

Give the class time to experiment beyond the base scenario.

- **Separable layout.** Cluster the white pieces in one quadrant and the black pieces elsewhere. Observe how the coefficients keep growing unless you cap them, highlighting why regularisation exists.
- **Ring layout.** Surround the white pieces with black pieces to demonstrate a non-linearly separable case. Prompt students to suggest extra features that would bend the boundary.

- **Train/test split.** Reserve a couple of pieces as a hold-out set and see how well the tuned model generalises.
- **Extra features in Orange.** Use **Formula** to add x^2, y^2, or $x \times y$ and compare the expanded model with the simpler baseline.

9.8 Stage 7: Rebuild the Workflow in Orange

Finish by connecting the physical intuition with a digital workflow.

1. Enter the chessboard coordinates into a small CSV file and load it with a **File** widget.
2. Use **Select Columns** to designate the colour as the target and the coordinates as features. Feed the data into a **Logistic Regression** widget.
3. Send the learner and data to **Predictions** so you can inspect the class probabilities beside the original labels.
4. Run the same data through **Test & Score**, then connect the results to **Confusion Matrix** and **Calibration Plot**. Compare the software predictions with the hand-worked threshold at 0.5.
5. Open **Scatter Plot** to colour the points by predicted class and discuss where the software boundary agrees with the board game and where it differs.

Capture photographs of the physical setup alongside screenshots of the Orange workflow so students can revisit the reasoning before the formal diagnostics chapter.

Chapter exercises

1. Record the chessboard coordinates and labels from the physical setup in a small table. Identify the feature columns and the target column.
2. Using supplied coefficients, compute the logit score and probability for three pieces. Show one calculation clearly.
3. Try one coefficient adjustment and decide whether it improves the hand-scored likelihood. Explain the trade-off if one piece improves while another gets worse.
4. Build confusion matrices at two different probability thresholds. Which threshold would you choose if false positives were more costly?
5. Rebuild the workflow in Orange and add one engineered feature such as x^2, y^2, or $x \times y$. Compare the expanded model with the simpler baseline.

Chapter 10

Logistic Regression and Model Diagnostics

Chapter overview

The earlier hands-on logistic-regression chapter gave these ideas a physical shape with chess pieces and a probability curve. This chapter now repeats the same story in Orange. We begin by showing how logistic regression can still overfit when we keep adding features or tuning against the same evaluation data, then use Orange's documented logistic-regression workflow to discuss regularisation, class probabilities, calibration, and threshold choice.

10.1 Guarding Against Overfitting

Even a simple classifier can become brittle when we add unnecessary features or keep retuning against the same evidence.

Learning objectives

After reading this section, you will be able to:

- explain why logistic regression can still overfit when the feature set balloons;
- use Orange to compare a simpler model with a more heavily engineered one;
- recognise when a validation gain is too small or unstable to trust.

Prerequisites

Before you begin, make sure you:

- understand the difference between training and validation data;
- can stage a classification dataset in Orange;

- know how to read cross-validation summaries from **Test & Score**.

Logistic regression is more constrained than a deep tree or a 1-NN boundary, but it is not immune to overfitting. The risk usually appears when we keep adding weak predictors, hand-crafted interaction terms, or squared features. Eventually the model starts explaining quirks that will not repeat outside the training sample. A second version of the problem comes from workflow habits: if we repeatedly tweak the same model after seeing the same validation table, we gradually overfit the evaluation process itself.

In Orange, a clean way to demonstrate this is to start with a modest feature set, then use **Formula** to add a few extra derived terms and compare both versions in **Test & Score**. If the expanded model lifts one fold but increases variation across folds, or if the gain is tiny compared with the noise, the simpler model is usually the safer choice. The point is not that feature engineering is bad. The point is that every new feature should earn its keep.

Controlled comparison.

1. Fit a baseline logistic model on the original features.
2. Add a small set of derived terms with **Formula** and refit the model.
3. Compare accuracy, F1, ROC AUC, and fold-to-fold spread in **Test & Score**.
4. Keep the expanded model only if the improvement is stable and interpretable.

Troubleshooting. If scores bounce around after adding features, inspect the staged data before drawing conclusions. Missing values, duplicated rows, or variables with almost no variation can make the comparison look worse than it really is.

Section summary

- Logistic regression can still overfit if we add too many weak or overly specific features.
- Stable validation gains matter more than one lucky fold.
- Feature expansion should be justified, not added just because the tool makes it easy.

Self-check questions

1. Why can a model with more engineered features perform worse on unseen data?
   Hint: Think about memorising quirks instead of learning the broader pattern.
2. What would convince you to keep the simpler feature set?
   Hint: Look beyond a tiny bump in one summary number.

10.2 Ridge and Lasso Regularisation

We use Orange's documented regularisation controls to keep logistic models stable as the feature set grows.

Learning objectives

After reading this section, you will be able to:

- configure Orange's **Logistic Regression** widget with L1 or L2 regularisation;
- explain how the cost parameter C changes the strength of regularisation;
- describe why stronger regularisation often improves generalisation.

Prerequisites

Ensure you can:

- explain the difference between L1 and L2 penalties in plain language;
- interpret cross-validation results;
- connect learners to **Test & Score**.

Orange's documented logistic-regression widget exposes the core controls we need here: the regularisation type and the cost parameter C. L2 regularisation usually shrinks many coefficients gently, while L1 pushes more aggressively toward sparse solutions. Lower values of C mean stronger regularisation. In practice, that means the model becomes less willing to chase every weak pattern in the training data.

Treat regularisation as a comparison, not a belief system. Run a small sweep such as $C \in \{1.0, 0.5, 0.1\}$ for both L1 and L2, then compare validation metrics. If the stronger penalty keeps the score similar while reducing fold-to-fold volatility, that is usually a win. The goal is not to produce the flashiest training fit. It is to choose a model that behaves sensibly on unseen cases.

Regularisation sweep.

1. Duplicate the **Logistic Regression** learner for a few C values.
2. Compare L1 and L2 under the same cross-validation setup.
3. Record which setting gives the best balance of score and stability.

Troubleshooting. If every regularised model performs badly, revisit the data pipeline before debating penalties. Poor staging, missing-value handling, or badly scaled numeric variables will overwhelm any benefit regularisation might offer.

Section summary

- Orange's logistic-regression widget gives us the essential regularisation controls: penalty type and C.
- Smaller C values apply stronger regularisation.
- The best setting is the one that improves generalisation, not merely the one that looks strongest on the training data.

Self-check questions

1. What does lowering C do to the model?
   Hint: Think in terms of how much freedom the coefficients are given.
2. Why might a slightly more regularised model be preferable even if its mean score is only marginally lower?
   Hint: Consider stability and reproducibility.

10.3 Classification Metrics and Calibration

We evaluate logistic models with metrics that actually match a classification task.

Learning objectives

By completing this section, you will be able to:

- distinguish accuracy, precision, recall, F1, ROC AUC, and calibration;
- explain why class imbalance can make accuracy misleading;
- connect metric choice to stakeholder priorities.

Prerequisites

Before continuing, ensure you:

- know how to interpret a confusion matrix;
- can run **Test & Score**;
- understand the idea of a predicted probability.

Once we move into classification, regression metrics stop being the main story. For a logistic model we care about whether the classes are separated well, whether the probability ranking is useful, and whether the chosen threshold matches the real-world trade-off. Accuracy gives a quick headline, but it can hide class imbalance. Precision tells us how

many positive predictions were correct. Recall tells us how many real positives we found. F1 balances those two when neither error type can be ignored. ROC AUC captures ranking quality across thresholds, while calibration asks a different question: when the model says 0.8, does that actually mean about an 80% chance?

These metrics are not interchangeable. A model can achieve respectable ROC AUC yet still be poorly calibrated, or it can post high accuracy by leaning on the majority class while missing most rare positives. That is why classification work needs a metric dashboard rather than a single favourite number.

Metric checklist.

1. Start with accuracy only as a rough summary.
2. Add precision, recall, and F1 when class imbalance matters.
3. Use ROC AUC to discuss ranking quality across thresholds.
4. Check calibration before treating predicted probabilities as decision support.

Troubleshooting. If one metric looks strong and another looks weak, do not assume the software is wrong. It usually means the model is good at one part of the task and weak at another, which is exactly what the dashboard is supposed to reveal.

Section summary

- Classification requires classification metrics; RMSE and R^2 are not the main tools here.
- Accuracy can be misleading when one class dominates.
- Calibration and ranking quality answer different questions and should both be checked.

Self-check questions

1. Why can accuracy be a poor guide on an imbalanced dataset?
   Hint: Imagine a model that simply predicts the common class every time.
2. What is the difference between a model that ranks well and a model that is well calibrated?
   Hint: One is about order, the other about probability meaning.

10.4 Modelling Log-Odds with Orange

We use Orange's documented workflow to move from class labels to probabilities, predictions, and calibration plots.

Learning objectives

After this section, you will be able to:

- prepare a binary classification problem in Orange without inventing extra encoding requirements;
- interpret logistic regression as a model of log-odds transformed into probabilities;
- connect **Logistic Regression**, **Predictions**, and **Calibration Plot** in one workflow.

Prerequisites

You should:

- understand the idea of odds and probabilities;
- know how to use **Select Columns** and **Continuize**;
- be familiar with Orange's basic learner workflow.

Orange does not require us to manually rename a binary target to 0 and 1 before using logistic regression. A two-class categorical target is enough. The important preparation step is not hand-encoding the class label; it is making sure the predictors are in a sensible numeric form when the chosen features require it. That is where **Continuize**, **Formula**, or careful staging in **Select Columns** matters.

Connect the staged data to **Logistic Regression**, then pass the result to **Predictions** and **Calibration Plot**. This keeps the interpretation grounded. The model learns a linear function on the log-odds scale; the logistic link converts that value into a probability between 0 and 1. **Predictions** shows the resulting class and probability outputs, while **Calibration Plot** lets us inspect whether those probabilities line up with observed outcomes.

Hands-on log-odds exploration.

1. Choose a binary target and stage the predictors carefully.
2. Fit **Logistic Regression** on the staged data.
3. Inspect class probabilities in **Predictions**.
4. Open **Calibration Plot** to compare predicted and observed probabilities.

Troubleshooting. If probability-based widgets report that no probabilities are available, check the workflow rather than the labels. The model, data roles, or evaluation connection is usually the problem, not the fact that the class values are words instead of numbers.

Section summary

- Orange logistic regression works with ordinary binary class labels.
- The key preparation task is getting the predictors into a suitable form.
- **Predictions** and **Calibration Plot** make the log-odds story concrete.

Self-check questions

1. Why is careful feature preparation more important here than manually forcing class labels to 0 and 1?
   Hint: Think about what the learner actually consumes.
2. Which Orange widget helps you check whether predicted probabilities behave sensibly?
   Hint: It compares predicted and observed class probabilities.

10.5 Logistic Loss in Plain Language

We build the conceptual background without pretending Orange exposes every optimisation detail.

Learning objectives

After working through this section, you will be able to:

- explain cross-entropy as a penalty for confident mistakes;
- describe why optimisation pushes the model toward probabilities that fit the data better;
- separate conceptual understanding from the limited controls exposed in a teaching GUI.

Prerequisites

Check that you:

- are comfortable with logarithms at a conceptual level;
- understand that training involves adjusting model parameters to reduce loss;
- do not expect every mathematical detail to appear as a widget setting.

The key loss function behind logistic regression is cross-entropy. It stays small when the model assigns high probability to the correct class and grows sharply when the model is confidently wrong. That makes it a natural training signal for a probabilistic classifier. In plain language, the optimiser keeps nudging the coefficients until those confident mistakes become rarer.

The important point here is conceptual rather than mechanical. Orange's logistic-regression widget does not function as a step-by-step optimisation visualiser, so we should not pretend otherwise. Use this section to understand what the model is trying to minimise, then rely on calibration and evaluation widgets to judge whether the fit is behaving sensibly.

A useful mental model. Think of training as repeatedly asking, "Did the current settings give low probability to cases that were actually positive, or high probability to cases that were actually negative?" If the answer is yes, the loss increases and the next parameter update tries to reduce that mistake pattern.

Section summary

- Cross-entropy punishes confident errors heavily.
- Logistic regression training searches for coefficients that reduce that penalty.
- In this chapter, optimisation is best taught conceptually rather than through undocumented GUI claims.

Self-check questions

1. Why is a confident wrong prediction worse than a hesitant wrong prediction under logistic loss?
   Hint: Think about what the loss function is trying to discourage.
2. Why should we avoid inventing Orange controls that are not part of the documented widget?
   Hint: Accuracy in the tool description matters as much as accuracy in the maths.

10.6 Classification Diagnostics

We use Orange's evaluation widgets to compare models, inspect operating trade-offs, and explain the result clearly.

Learning objectives

By the end of this section, you will be able to:

- use **Confusion Matrix**, **ROC Analysis**, and **Calibration Plot** to evaluate a logistic model;
- explain how threshold changes move precision, recall, and error rates;
- tie the chosen operating point back to stakeholder costs.

Prerequisites

Before proceeding, make sure you:

- configured at least one logistic model and one comparison learner;
- can read confusion matrices and ROC curves;
- understand the consequences of false positives and false negatives in the domain.

The diagnostic workflow begins with **Test & Score**. Send the evaluation results to **Confusion Matrix** for class counts, **ROC Analysis** for ranking comparisons, and **Calibration Plot** to check whether the predicted probabilities are trustworthy. Each widget answers a different question. The confusion matrix tells you what happened at one operating point. ROC analysis compares how well the models rank cases before the operating point is fixed. Calibration Plot asks a different question: when the model says 0.8, do cases with that score really come out positive about 80% of the time?

This is also the right place to connect metrics to real costs. A medical screening model may prefer higher recall even if false positives rise. A marketing model may accept lower recall to protect precision. Orange helps us compare the ranking and inspect the current operating point, but if we need a formal threshold sweep with cost tables, we export the probabilities and evaluate those candidate thresholds explicitly instead of guessing from a screenshot.

Diagnostic workflow.

1. Run the learners through **Test & Score**.
2. Open **ROC Analysis** to compare ranking strength across models.
3. Open **Confusion Matrix** to inspect the current operating point.
4. Use **Calibration Plot** to check whether the predicted probabilities are well aligned with observed outcomes.
5. If the task needs a non-default threshold, export the probabilities and compare a small set of candidate cut-offs in a documented table.

Troubleshooting. If the ROC view is uninformative or the confusion matrix looks lopsided, check class balance and the selected target class first. Diagnostic confusion often comes from a mismatch between the modelling question and the metric being emphasised.

Section summary

- Different Orange evaluation widgets answer different questions about the same classifier.
- Threshold choice should be deliberate and tied to the cost of errors.
- Calibration is about probability quality, not just about choosing a cut-off.

Self-check questions

1. Which Orange widget would you use to compare model ranking before locking in a threshold?
   Hint: Think about curve comparisons rather than a single confusion matrix.
2. Which Orange widget helps you check whether predicted probabilities mean what they seem to mean?
   Hint: It compares predicted probabilities with observed frequencies.

Conclusion and next steps

We now have a cleaner picture of logistic regression in Orange: use the documented regularisation controls, evaluate it with classification metrics, check calibration, and choose thresholds on purpose. Carry those habits into Hands-on Activity 3. The chessboard exercise is simple on purpose, but it should reinforce the same probability, threshold, and diagnostic thinking that the software workflow introduced here.

Chapter exercises

1. Identify three warning signs that a modelling workflow is overfitting either the training data or the evaluation process itself.
2. Compare a small coefficient table before and after L1 or L2 regularisation. Which features were shrunk most, and what does that suggest?
3. Given a confusion matrix, compute accuracy, precision, recall, and F1. Then choose which metric should lead the discussion for the scenario.
4. Interpret a small calibration table. If the model assigns probability 0.8 to a group of cases, what would good calibration look like?

5. Build or inspect an Orange diagnostic workflow using **Test & Score**, **Confusion Matrix**, **ROC Analysis**, and **Calibration Plot**. Write a short recommendation from the evidence.

Chapter 11

Understanding and Tuning k-NN

Chapter overview

The nearest-neighbour family gives us a complementary perspective to logistic regression. We begin by anchoring the voting intuition, move through distance metrics and scaling choices, and finish by tuning hyperparameters with evaluation safeguards.

These sections set up the NSW land value and Red Rooster case studies that follow, and they foreshadow the comparisons we make in Hands-on Activity 4 when we place neighbour and logistic boundaries side by side.

11.1 Voting Intuition and Decision Boundaries

We start with the voting intuition, see how the decision boundary responds to different k, and meet the practical dials we can adjust.

Learning objectives

After reading this section, you will be able to:

- describe k-NN as a majority-voting classifier that assigns new examples the label held by nearby training points;
- explain how varying k alters decision boundary smoothness and the bias–variance trade-off;
- sketch decision regions in Orange to support the land-value case study in chapter 12.

Prerequisites

Before you begin, make sure you can:

- read Orange scatter plots that colour-code class labels;
- interpret confusion matrices and class proportions;

- distinguish between training and evaluation data splits.

k-NN treats every labelled example as a voting neighbour. When a new parcel or customer arrives, we measure its distance to the existing records and tally the labels of the closest k points.

The majority outcome becomes the prediction, so the model's behaviour lives entirely in the geometry of the training set rather than in fitted coefficients.

Visualising the decision boundary—the curve that separates predicted classes—helps us see how the vote changes as we move through feature space. With $k = 1$, the boundary hugs each observed point, weaving jagged islands around every example.

Raising k makes the regions smoother by insisting that a larger neighbourhood agree before we switch classes, trading local sensitivity for stability.

The Orange workflow mirrors that logic. Connect a staged dataset to **Scatter Plot** and toggle the "Show decision boundary" option. Slide the k parameter in the linked **k Nearest Neighbours** widget between 1, 3, 5, and 15 to watch the coloured regions inflate or contract.

Capture a short note about how the boundary changes as k changes. That observation becomes useful later when we compare neighbours with logistic regression on the same layout.

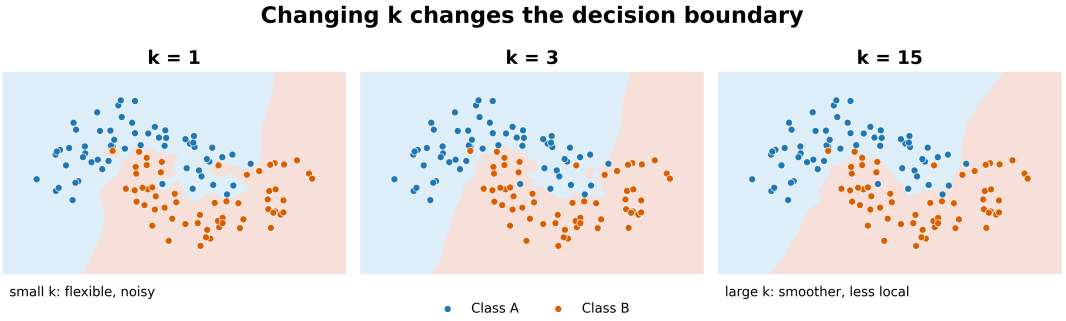


Figure 11.1: A small k lets individual training points carve jagged regions, while a larger k smooths the boundary by requiring a broader neighbourhood to agree.

Accessibility spotlight. Press `Ctrl+F1` in Orange to open widget help without leaving the keyboard, and use `Tab` to step through parameter fields.

These shortcuts keep the demonstration inclusive for learners who rely on keyboard navigation.

Try this variation. Load the Palmer Penguins dataset from the practice bundle (`resources/`). Plot bill length against flipper length to see how the Gentoo decision region expands as you increase k.

Troubleshooting. If the decision boundary looks noisy even for high k, confirm that the target variable is categorical in **Select Columns**. Numeric targets trigger regression mode,

which draws a continuous surface instead of class regions.

Section summary

- k-NN predicts labels by letting nearby observations vote on the outcome.
- Small k values create flexible but fragile boundaries; larger k smooths away noise.
- Visual boundary checks in Orange prepare us for the land-value workflow in chapter 12.

Self-check questions

1. How would you explain the trade-off between $k = 1$ and $k = 15$ to a stakeholder?
   Hint: Focus on how much of the neighbourhood must agree before the vote flips.
   Common misconception: Higher k always improves performance.
   If you answered "pick the largest k," revisit how overly smooth boundaries lose accuracy.
2. Which Orange widgets help you visualise decision regions while adjusting k?
   Hint: Think about the combination of model and plot.
   *Common misconception: The **Scatter Plot** works alone.*
   *If you answered "just Scatter Plot," remember that the **k-NN** widget must feed it a model first.*

11.2 Configure Distance Metrics, Weighting, and Scaling

We tune the practical dials that shape k-NN behaviour: metric choice, neighbour weighting, and feature scaling.

Learning objectives

After reading this section, you will be able to:

- configure Orange's k-NN widget for classification or regression scenarios;
- compare uniform and distance-weighted voting schemes;
- normalise features so distance metrics treat each scale fairly.

Prerequisites

Ensure you can:

- interpret Orange's **Column Statistics** output;
- compute simple min–max and z-score transformations;

- explain the difference between Euclidean and Manhattan distance.

k-NN adapts to multiple task types. When the target is categorical, neighbours vote. When the target is numeric, the learner behaves as a regressor and averages nearby target values instead. This behaviour supports our NSW land-value estimate: the predicted price per square metre becomes the mean of the surrounding parcels, optionally weighted by distance when nearby blocks should influence the result more strongly than far-flung ones.

Distance weighting also matters for classification. Uniform weighting treats every neighbour equally, whereas inverse-distance weighting gives closer observations a louder voice, preventing fringe cases from dominating the vote.

We must also choose an appropriate distance metric. Euclidean distance measures straight-line separation, suiting continuous coordinates, while Manhattan distance sums absolute differences along each axis. Regardless of the metric, unscaled features can derail the model: a single wide-ranging attribute such as "land area" can swamp others like "zoning code." Use **Normalize** to put numeric features on a common scale, and **Continuize** to encode categorical predictors as numbers, before fitting. Document your choices in the experiment log so readers of chapter 13 can reproduce the comparisons.

Workflow checklist.

1. Inspect feature ranges with **Column Statistics** and note any dominant scales.
2. Apply **Normalize** (z-score or min–max scaling) so no numeric feature dominates the distance calculation, and use **Continuize** to encode any categorical features as numbers.
3. Configure k **Nearest Neighbours** with the chosen metric, weighting, and k value.
4. Record the configuration and rationale in your Orange annotations.

Try this variation. Experiment with Manhattan distance on the Red Rooster case study to see how axis-aligned neighbourhoods reshape the decision boundary.

Troubleshooting. If Orange reports "No numeric features available," check that categorical attributes were one-hot encoded via **Continuize**. For regression, confirm that the target remains numeric; accidental type conversion will hide it from the widget.

Section summary

- k-NN can vote for classes or average numeric targets depending on the type of target column.
- Distance weighting and metric selection tailor the neighbourhood to the domain.
- Normalising features keeps the distance calculation balanced across scales.

Self-check questions

1. When would you prefer inverse-distance weighting over uniform voting?
   Hint: Think about how similar the nearby cases are.
   Common misconception: Weighting only matters for regression.
   If you answered "never," revisit the geospatial parcel example where distant suburbs distorted votes.

2. How do you prepare categorical zoning codes for a k-NN model?
   Hint: Consider the encoding step.
   Common misconception: Leave them as strings.
   If you answered "k-NN can read text directly," remember that distance metrics need numbers.

11.3 Operational Trade-offs

We weigh the implementation realities of k-NN: storage, privacy, latency, and deterministic outcomes.

Learning objectives

After reading this section, you will be able to:

- articulate the cost of storing full training sets for k-NN deployments;
- design tie-breaking policies that keep predictions reproducible;
- plan query-time optimisation strategies for latency-sensitive products.

Prerequisites

Before continuing, ensure you can:

- estimate storage requirements for medium-sized tabular datasets;
- describe basic indexing structures such as KD-trees or ball trees;
- communicate privacy obligations for customer data.

Deploying k-NN means shipping the entire training dataset, because prediction requires comparing each new query against stored examples. That obligation complicates privacy compliance: sensitive attributes remain at rest, so we must encrypt disks, manage retention policies, and justify why keeping raw records is necessary. Storage also affects latency. Without indexing, each prediction scans every row, slowing down mobile experiences. Pre-computing spatial indices or caching frequent queries helps the model respond within practical limits.

Determinism deserves equal attention. When two classes receive the same number of votes, a tie-break rule chooses the winner. Document whether you default to the majority class, the closest neighbour, or a weighted decision so stakeholders can reproduce the result. Tie policies matter in civic applications like valuation, where consistent outputs build trust.

Workflow checklist.

1. Estimate memory usage by multiplying the number of records by the feature count and storage size.
2. Choose an indexing strategy (KD-tree, ball tree) if strict latency bounds apply.
3. Define and document tie-break rules in the project README.
4. Review privacy controls with stakeholders before deploying beyond experimentation.

Troubleshooting. If Orange predictions feel slow on large datasets, reduce k temporarily or sample the training set while you evaluate feasibility. For production builds, move the workflow into Python with `KNeighborsClassifier` from `sklearn.neighbors` so you can take advantage of compiled indices.

Section summary

- k-NN stores the training set, so privacy and latency must be planned up front.
- Tie-breaking policies keep outcomes reproducible for stakeholders.
- Query-time optimisation ensures the method remains viable in real products.

Self-check questions

1. What documentation should accompany a deployed k-NN model to explain how ties are resolved?
   Hint: Consider reproducibility and stakeholder trust.
   Common misconception: Ties are so rare that no policy is needed.
   If you answered "none," revisit how evenly split parcels in the land-value dataset force explicit choices.
2. How would you address a product manager's concern that storing the training set breaches privacy agreements?
   Hint: Think about anonymisation, encryption, and retention limits.
   Common misconception: Deleting identifiers is sufficient.
   If you answered "strip the IDs," remember that the attribute mix can still identify households, so governance must be broader.

Conclusion and next steps

We have covered the voting intuition, distance metrics and weighting, feature scaling, and hyperparameter tuning with evaluation safeguards. The NSW land value and Red Rooster case studies reuse these settings to interpret regional price gradients and restaurant site predictions, and Hands-on Activity 4 places these neighbour models side by side with logistic baselines.

Chapter exercises

1. Compute the distances from one query point to five labelled points and predict the query label using 1-NN and 3-NN.
2. Repeat the calculation after rescaling one feature. Which neighbour changed position in the ranking, and why?
3. Compare validation results for $k = 1$, $k = 3$, $k = 7$, and $k = 15$. Choose a value of k and justify it using both performance and stability.
4. Explain when inverse-distance weighting is useful and when it might amplify noise.
5. Write a short operations note naming one risk that would make a deployed k-NN model stale over time.

Chapter 12

Predicting NSW Land Value in Orange

Chapter overview

This case study applies the k-NN principles to a real regional dataset. We explore the valuation records, engineer features that respect spatial context, and evaluate neighbour regressors alongside baselines. Each stage prepares us to compare land value predictions with the restaurant classification story and with later tree and forest regressors.

12.1 Understand the Valuation Data

We work through the tax-oriented valuations, derived features, and spatial quirks of the dataset.

Learning objectives

After reading this section, you will be able to:

- summarise unimproved land value records and the variables captured by the NSW government dataset;
- derive price-per-square-metre features that make neighbouring parcels comparable;
- identify missing or unreliable valuations that require cautious interpretation.

Prerequisites

Before diving in, make sure you can:

- interpret government valuation metadata and zoning codes;
- calculate ratios and unit conversions for tabular data;

- navigate Orange's **File** and **Select Columns** widgets.

The NSW Valuer General releases "unimproved land value" assessments that capture what a parcel would sell for without considering buildings. Each record includes identifiers, geographic coordinates, lot sizes, and recent valuation amounts. We start by reading the CSV into Orange and checking the schema: lot area sits in square metres, while valuations appear as whole-dollar amounts tied to specific reporting years. Computing a price-per-square-metre feature by dividing the valuation by lot area lets us compare irregular parcels on a common footing.

Source note. This chapter uses a course-prepared sample derived from NSW land-value records published through Property NSW and the Valuer General. The textbook keeps only the fields needed for the classroom exercise; see the references chapter for the official land-value source.

Bushland and recreational reserves often lack reliable valuations, either because they are exempt from rating or their usage makes open-market comparisons tricky. Our workflow focuses on predicting an unseen reserve's value by leveraging nearby residential and mixed-use parcels. Highlight the "missing valuation" rows in **Data Table** and note which suburbs cluster around the gap so we can validate predictions later.

Because the coordinates arrive as latitude and longitude, we also need to be honest about distance. Raw Euclidean distance on angular coordinates is only an approximation. For a local classroom exercise within one region of NSW, a simple planar adjustment can be acceptable, such as scaling longitude by $\cos(\varphi)$ for the local latitude before computing neighbour distances. For a larger area or a production workflow, projected coordinates or a proper geographic distance measure are the safer choice.

Section summary

- NSW valuation data records unimproved land values alongside lot size and coordinates.
- Deriving price-per-square-metre normalises parcels of different sizes.
- Missing valuations motivate predicting a bushland reserve using nearby parcels.

Self-check questions

1. Which attributes do you need to compute price per square metre, and why is this ratio helpful?
   Hint: Think about comparing parcels of different sizes.
2. How would you document the parts of the dataset that lack valuations?
   Hint: Consider annotations in Orange and comments in your lab notes.

12.2 Build the Orange Valuation Workflow

We translate the valuation brief into a reproducible Orange canvas that keeps the query parcel separate from the training data.

Learning objectives

After reading this section, you will be able to:

- build the baseline Orange workflow that feeds staged valuations into a k-NN regressor;
- create a synthetic query instance without contaminating the training data;
- log intermediate checks so future readers can reproduce the canvas.

Prerequisites

Ensure you can:

- use **Select Columns** to mark targets and metas;
- operate Orange's **Create Instance** widget;
- interpret Orange annotations and save workflows for later reuse.

Start with **File** to load the valuation CSV, then pass it to **Select Columns**. Mark "price_per_sqm" (the derived feature) as the target, keep the adjusted spatial coordinates as features, and treat identifiers as meta fields. Use **Formula** to create any derived columns you need, such as price per square metre or a longitude adjustment for local distance calculations. Add **Column Statistics** to confirm that ranges look sensible and to catch any zero-valued lots that might distort averages.

Next, connect **Create Instance**. Enter the coordinates of the bushland parcel we want to value, along with its lot size if available. Disable "Append instance to data" so the query remains separate. Finally, attach k **Nearest Neighbours** with inverse-distance weighting. Because the target is numeric, the learner behaves as a regressor in this workflow. Route both the original data and the query into **Predictions**, then save the workflow with your lab notes so teams can revisit the configuration during the hands-on session.

Workflow checklist.

1. Load the staged CSV via **File** and inspect types in **Select Columns**.
2. Derive price per square metre and any local coordinate adjustments using **Formula**.
3. Configure **Create Instance** with the query parcel and disable appending.
4. Connect k **Nearest Neighbours** with inverse-distance weighting and confirm that the numeric target is set correctly.
5. Send outputs to **Predictions** and **Data Table** for inspection.

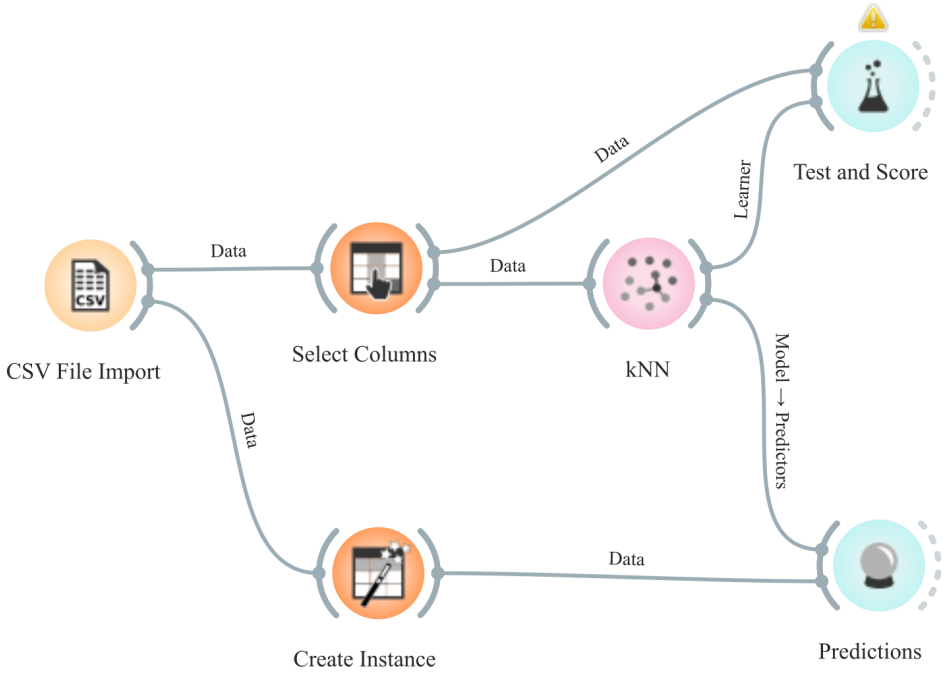


Figure 12.1: Orange workflow for the NSW land-value case study. The query parcel is created separately and sent to **Predictions** alongside the fitted k-NN regressor, while **Test & Score** evaluates the same learner on known parcels.

Troubleshooting. If **Create Instance** refuses to generate a prediction, check that you supplied every feature the model expects. Missing coordinates will trigger "No data" warnings until the field is filled.

Try this variation. Duplicate the workflow and replace k-NN with **Random Forest**. Compare the predicted valuation and explain when a tree-based model might outperform distance-weighted averages.

Section summary

- The Orange canvas combines **File**, **Select Columns**, **Create Instance**, and *k* **Nearest Neighbours** for valuation.
- Disabling instance appending keeps the query parcel out of the training data.
- Annotated workflows document the modelling choices for future labs.

Self-check questions

1. Why do we disable "Append instance to data" when valuing the bushland parcel? *Hint: Think about data leakage.*

2. Which widget confirms that the feature ranges look sensible before modelling?
   Hint: Recall the diagnostic you inserted after ***Select Columns****.*

12.3 Interrogate the Prediction

We validate on known parcels, run sensitivity analysis, and apply sanity checks before trusting a single estimate.

Learning objectives

After reading this section, you will be able to:

- validate the model by predicting on held-out parcels with known valuations;
- assess sensitivity to different k values and distance weightings;
- document follow-up checks for geocoding accuracy and policy caps.

Prerequisites

Before continuing, make sure you can:

- perform train/test splits in Orange;
- read **Test & Score** reports for regression metrics;
- investigate geospatial artefacts such as duplicated coordinates.

Hold back a subset of parcels with known valuations and feed them through **Test & Score** alongside the query instance. Comparing root-mean-square error (RMSE) and mean absolute error (MAE) across different k values helps us understand whether the model over-smooths neighbourhood effects. If increasing k inflates RMSE, we know that our bushland prediction might be similarly dampened.

Sensitivity analysis continues the story. Duplicate the k **Nearest Neighbours** widget with $k = 3$, $k = 7$, and $k = 15$, then log how the bushland estimate changes. Large swings suggest the area lacks dense comparables, hinting that we should collect more local sales data or consult planners for context. Finally, audit the geocoding: plot the training and query points on **Geo Map** or export coordinates for a quick check in a GIS tool. Regional policy caps, such as limits on waterfront valuations, can flatten predictions; note these constraints in the commentary so stakeholders understand when manual review is required.

Troubleshooting. If **Test & Score** shows "NaN" metrics, ensure that every evaluation fold contains at least one example—sparse rural areas may require stratified sampling or manual grouping before splitting.

Try this variation. Evaluate the workflow using the Singapore resale-flat sample from the practice bundle. Discuss how the valuation features differ and which preprocessing steps must change for international datasets.

Document the evidence. Capture screenshots of the evaluation metrics, sensitivity table, and map view. Store them with descriptive filenames (for example, `nsw-valuation-k3-rmse.png`) so future cohorts can trace the reasoning behind the recommended k.

Section summary

- Validating on known parcels builds trust before predicting new valuations.
- Sensitivity checks across k values reveal how stable the bushland estimate is.
- Geocoding audits and policy notes contextualise the numerical output.

Self-check questions

1. How do you decide whether the chosen k produces an acceptable valuation error?
   Hint: Compare RMSE and MAE across multiple settings.
2. What additional evidence would you gather if the prediction changed dramatically between $k = 3$ and $k = 15$?
   Hint: Think about map inspections and stakeholder input.

Conclusion and next steps

By grounding k-NN in NSW valuation data we have practised feature engineering, spatial reasoning, and clear reporting of evaluation results that align with real policy questions. We can keep the annotated canvas handy as we compare these neighbour regressors with tree and random-forest regressors later. The contrasts between regression and classification also sharpen our instincts when we revisit the Red Rooster case study elsewhere in the book.

Chapter exercises

1. Prepare a data-role audit for the land-value sample. Mark each field as target, feature, metadata, or excluded, and explain two choices.
2. Build the Orange valuation workflow with **Create Instance**. Explain why the query parcel should not be appended to the training data.

3. Run a sensitivity table for at least three values of k and two weighting choices. Record how much the predicted value moves.

4. Compare RMSE and MAE on known parcels. Decide whether the query prediction is stable enough to report and what caveat belongs beside it.

5. Name two non-model checks a planner or valuer should make before relying on the estimate.

Chapter 13

Red Rooster Line: Choosing the Right Classifier

Chapter overview

This chapter extends the k-NN discussion by contrasting it with logistic regression on a geospatial classification task. We set up the Red Rooster line problem, compare decision boundaries, and reflect on operational considerations when rolling out the better-performing model. The sections set up the evaluation guard rails and the Hands-on Activity 4 comparison that follow.

13.1 Frame the Geospatial Challenge

We use the chicken-shop case study to contrast logistic regression with k-NN, showing how geography influences algorithm choice.

Learning objectives

After reading this section, you will be able to:

- describe the "Red Rooster line" as a supervised classification problem over latitude and longitude;
- prepare the fast-food dataset by filtering the relevant chains and constructing training/test splits;
- articulate why regional imbalance matters when comparing classifiers.

Prerequisites

Before starting, ensure you can:

- stage datasets in Orange using **File** and **Select Columns**;

- interpret class imbalance metrics such as prevalence and lift;
- explain logistic regression and k-NN basics from chapter 11.

The folklore surrounding the "Red Rooster line" suggests that one fast-food chain dominates north of Sydney while another prevails south of a notional boundary. We model this story as a binary classification problem: each store location has latitude, longitude, and a label indicating the franchise. The first step is to filter the dataset so only the chains of interest remain, preserving store counts for both classes. Record the prevalence—the proportion of stores belonging to each chain—because imbalance shapes how we judge model performance.

Source note. The teaching dataset for this chapter is a filtered, geocoded copy of NSW fast-food chain locations prepared for the course materials. Later extensions in the course enrich it with ABS neighbourhood covariates; see the references chapter for the official census and software sources behind that workflow.

Split the data into training and testing subsets using **Data Sampler** or **Data Sampler (Fixed)** to keep evaluation consistent. Tag the splits in your notes so you can reproduce them when iterating on parameters. This disciplined setup paves the way for the side-by-side comparison later in the chapter.

Accessibility spotlight. Label major geographic landmarks or directions clearly when you present the scatter plot so the spatial story is easy to follow.

Section summary

- The Red Rooster line becomes a latitude/longitude classification task with two chains.
- Filtering and documenting class prevalence prepares the data for fair comparisons.
- Consistent train/test splits make subsequent evaluation trustworthy.

Self-check questions

1. How does class imbalance influence your expectations for accuracy in this case study?
   Hint: Consider what happens if one chain heavily outnumbers the other.
2. Which Orange widgets help you capture reproducible train/test splits?
   Hint: Think about sampling tools.

13.2 Compare Logistic Regression with k-NN

We build a fair side-by-side comparison and explain why local geography can favour a more flexible boundary.

Learning objectives

After reading this section, you will be able to:

- build an Orange canvas that evaluates 1-NN, 3-NN, and logistic regression side by side;
- diagnose when logistic regression oversimplifies the decision boundary;
- prepare the feature space (for example by scaling longitude) before comparing metrics.

Prerequisites

Ensure you can:

- interpret confusion matrices and ROC curves;
- scale features using **Normalize** or **Continuize**;
- route models into **Test & Score** and **Confusion Matrix** widgets.

Start by correcting the coordinates for geography. Because a degree of longitude is shorter than a degree of latitude around Sydney, a quick fix is to multiply longitude by roughly $\cos(\varphi)$ for the local latitude (about 0.84 here) before comparing distances. Feed the adjusted data into three models: k **Nearest Neighbours** with $k = 1$ and $k = 3$, plus **Logistic Regression**. Connect each to **Test & Score** and **Confusion Matrix** so you can inspect accuracy, F1, ROC AUC, and misclassification patterns.

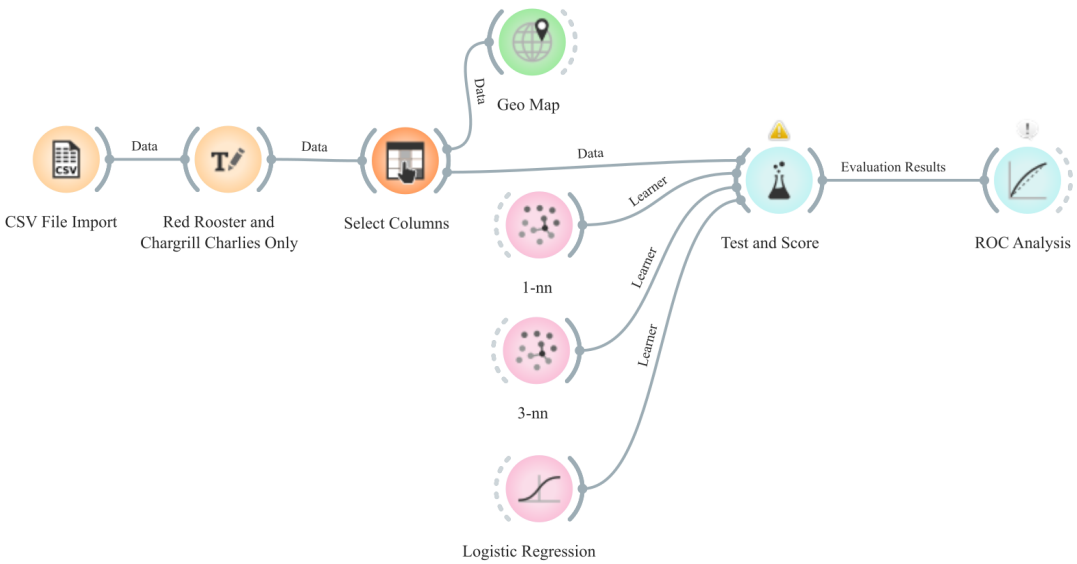


Figure 13.1: Orange workflow for the Red Rooster comparison. The filtered store data feeds a map view and three competing learners, with the shared **Test & Score** output routed to **ROC Analysis** for ranking diagnostics.

As you explore the results, examine the scatter plot with decision regions enabled. Logistic regression draws a single linear boundary that sometimes defaults to predicting the dominant chain. When the real boundary snakes along the harbour, k-NN retains the regional quirks by letting local neighbours steer the prediction. Capture screenshots of each model's boundary, labelling them clearly for learners reviewing chapter 11.

Try this variation. Swap the target to "Is the store within 1 km of a highway?" and compare how the models behave when the boundary becomes less linear. This encourages readers to match model flexibility to spatial structure.

Troubleshooting. If logistic regression outputs identical probabilities for every point, check that feature scaling applied correctly and that the class labels were staged as intended. For k-NN, confirm that you reconnected the scaled data rather than the raw coordinates.

Section summary

- Scaling coordinates keeps logistic regression and k-NN comparisons fair.
- Logistic regression provides a smooth baseline, while k-NN captures local variation.
- Visualising decision regions alongside metrics grounds the model choice in evidence.

Self-check questions

1. Which metrics beyond accuracy influenced your choice between logistic regression and k-NN?
   Hint: Think about ROC AUC, F1, or MCC.
2. How does scaling longitude alter the confusion matrices?
   Hint: Compare the false positive rates before and after scaling.

13.3 Interpret the Results

We weigh the accuracy swings and the final model choice through the metrics.

Learning objectives

After reading this section, you will be able to:

- evaluate variance in model accuracy using repeated sampling or cross-validation;
- communicate findings using metrics that resonate with stakeholders;
- justify the final model choice with evidence collected during the comparison.

Prerequisites

Before proceeding, make sure you can:

- run **Test & Score** with cross-validation or repeated sampling;
- interpret confidence intervals for accuracy and ROC AUC;
- present findings in meeting-ready notes or dashboards.

Use repeated cross-validation in **Test & Score** to estimate variability. Record the mean and standard deviation for accuracy and ROC AUC across folds. If logistic regression shows low variance but systematically underperforms k-NN, you can argue that its stability does not compensate for the accuracy gap.

Translate the findings into stakeholder-friendly language. Highlight metrics that connect to business questions: F1 captures balance between false positives and negatives, ROC AUC summarises ranking quality, and Matthews correlation coefficient (MCC) handles imbalance gracefully. Present the numbers with context, noting that a 3-NN model improves accuracy by a measurable margin while keeping interpretation grounded in the local geography.

Finally, recommend the model with a concise justification. For example: "We propose 3-NN because it increases ROC AUC by 0.07 relative to logistic regression and better respects the harbour boundary, as shown in the figure above." Include the saved workflow and the filtered data table so readers can replicate the analysis without rebuilding it from scratch.

Document the decision. Add a short write-up to your lab log summarising the evaluation settings, chosen metrics, and final recommendation. This practice mirrors the reporting expectations in chapter 12 and Hands-on Activity 4.

Troubleshooting. If cross-validation results vary wildly between runs, ensure that the random seed in **Data Sampler (Fixed)** matches across experiments. Consistent seeding keeps comparisons fair.

Section summary

- Repeated sampling reveals whether observed accuracy differences are stable.
- Communicating metrics in plain language helps stakeholders follow the argument.
- The 3-NN model often wins because it respects geographic nuances that logistic regression flattens.

Self-check questions

1. Which evidence would you include in a one-page summary recommending 3-NN over logistic regression?
   Hint: Combine metrics, visuals, and workflow references.

2. How do you reassure stakeholders that the evaluation will hold if the dataset grows?
   Hint: Discuss cross-validation, monitoring, and periodic retraining.

Conclusion and next steps

The Red Rooster case compared classifiers on a geospatial task, balancing accuracy with geographic nuance and operational constraints. As we move into the evaluation guard rails chapter, we carry forward our notes on threshold choices, refresh schedules, and stakeholder messaging. They feed into Hands-on Activity 4, the classifier-comparison activity that makes these contrasts concrete.

Chapter exercises

1. Apply the longitude scaling idea to three sample store coordinates. Explain why the result is useful locally but not a full geographic projection.

2. Build or inspect the Orange workflow comparing 1-NN, 3-NN, and logistic regression. Save the workflow screenshot and identify the shared evaluation node.

3. Use **Test & Score** and confusion matrices to choose a model. Do not rely on accuracy alone; mention at least one ranking or balance metric.

4. Compare decision-region screenshots for logistic regression and k-NN. What geographic assumption does each model make?

5. Write a one-page recommendation that includes metrics, a visual reference, a limitation, and a suggested refresh schedule.

Chapter 14

Evaluate Models with Guard Rails and Baselines

Chapter overview

With k-NN and logistic classifiers fresh in mind, this chapter consolidates our evaluation discipline. We stabilise metrics with cross-validation, guard against the garden of forking paths, and compare models against meaningful baselines. These guard rails directly support the Red Rooster case study in this part of the book and the later land-value regression extension, and they lead into Hands-on Activity 4, where we debate classifier choices using the same evidence.

14.1 Stabilise Metrics with Cross-Validation

Detail validation schemes that temper noisy train/test splits and inform hyperparameter choices.

Learning objectives

After reading this section, you will be able to:

- configure Orange's **Test & Score** widget for k-fold validation with reproducible seeds;
- interpret the mean and variance reported for each metric across folds;
- justify when to reserve a sealed hold-out set in addition to cross-validation.

Prerequisites

Before starting, make sure you can:

- load datasets and connect learners within the Orange canvas;

- explain the difference between training, validation, and test data partitions;
- calculate averages and standard deviations for small tables of results.

Orange's **Test & Score** widget gives us a reproducible way to average over several train/test splits without juggling multiple canvases. Instead of trusting a single random partition, we select k-fold cross-validation. This scheme slices the data into k equally sized folds and rotates the hold-out fold, then reports the mean and variance of each metric across the rotations. The average smooths away unlucky splits that would otherwise produce contradictory accuracy scores, while the spread tells us when a model is inherently unstable. We keep baselines, tree ensembles, and neighbours in the same Test & Score run so we can compare them on identical folds rather than on separate experiments that might favour one model by chance.

Once we have the cross-validated summary, we still preserve a final hold-out partition for the end of the project. The hold-out data remains sealed until we commit to a modelling recipe, ensuring the reported test metrics reflect how the system will behave on new, unseen cases. This separation mirrors the staged evaluation in the logistic regression diagnostics chapter (chapter 10) and prepares us for production roll-outs where the first real-world batch effectively acts as another uncompromised test.

Worked example: auditing franchise scores. Figure 14.1 shows a representative evaluation branch for the Red Rooster franchise dataset: preprocessing feeds a random forest, **Parameter Fitter**, feature-importance checks, and the shared evaluation widgets. In practice, logistic regression, k-NN, and a constant baseline would also connect to the same **Test & Score** node so every learner is judged on identical folds. We select $k = 10$, fix the random seed so teammates can reproduce the splits, and compare accuracy, ROC AUC, F1, MCC, and log loss across folds. The mean ROC AUC highlights that the random forest ranks locations best, while the standard deviation warns that k-NN swings wildly across folds. We log these observations in the project journal before touching the hold-out set.

Troubleshooting checklist

- If Orange refuses to run cross-validation, confirm every learner receives the same preprocessed data stream and that categorical features are encoded consistently.
- When metrics disagree across folds, inspect the **Data Table** output from each fold to catch class imbalance or duplicate records creeping into the splits.
- If results differ between team members, verify the random seeds in **Test & Score** and **Data Sampler** match the documented configuration.

Computing extension. If you want nested cross-validation, move the experiment into Python rather than forcing Orange to behave like a full tuning

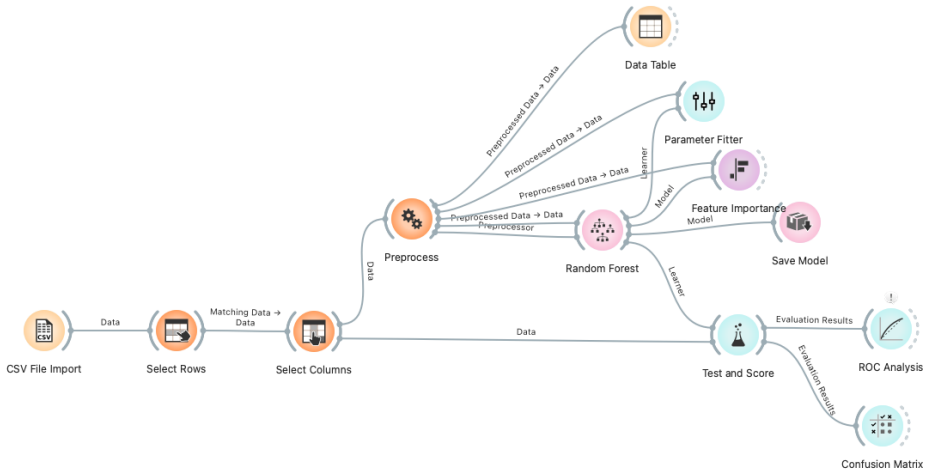


Figure 14.1: Representative Orange evaluation workflow for the Red Rooster dataset, showing preprocessing, a random forest branch, parameter tuning, feature-importance inspection, and shared evaluation widgets. Additional learners can be connected to the same **Test & Score** node for fair comparison.

framework. Orange is excellent for small, transparent comparisons; script-based tooling is better once you need nested loops or richer search logic.

Section summary

- Cross-validation averages metrics over multiple folds, smoothing unlucky splits while exposing model instability.
- Reserving a sealed hold-out set protects the final evaluation from bias introduced during model selection.
- Documented seeds and shared workflows keep team members aligned when they revisit an experiment.

Self-check questions

1. Why do we still protect a hold-out set even after running k-fold cross-validation?
2. Which symptoms tell us a learner may be unstable across folds, and how should we respond?

14.2 Guard Against the Garden of Forking Paths

Warn readers about over-searching hyperparameters and reusing evaluation data.

Learning objectives

After reading this section, you will be able to:

- set boundaries on hyperparameter grids before launching a tuning sweep;
- describe logging practices that keep evaluation steps auditable;
- configure Orange's widgets so validation and testing partitions remain isolated.

Prerequisites

Before starting, ensure you:

- understand how **Data Sampler** partitions datasets;
- can export CSV logs from Orange widgets for record keeping;
- are comfortable interpreting standard error values reported by **Test & Score**.

Cross-validation is only trustworthy when we resist the temptation to rerun it indefinitely. Sweeping dozens of k values or decision-tree depths invites the "garden of forking paths": the more dials we try, the higher the chance that one combination wins by accident. To keep our search disciplined, we set expectations up front: choose a modest grid of plausible hyperparameters, log every candidate, and stop once the cross-validated difference between contenders is smaller than the reported standard error. When two models tie, we prefer the simpler configuration or whichever one is easier to explain in the accompanying hands-on activity.

Validation data should never blend with the final testing pass. Orange makes this easier by letting us allocate a separate test split in **Data Sampler** and send only the training portion through hyperparameter tuning widgets. We document the split ratios, random seeds, and selection rules in our project notes so teammates cannot accidentally reuse the test fold while iterating. This audit trail protects us when stakeholders ask how confident we are in the reported improvements, and it gives future analysts enough context to reproduce or extend the experiment without leaking information.

Worked example: disciplined parameter sweeps. We build on the workflow from Figure 14.1 by connecting **Data Sampler** ahead of **Test & Score**. The sampler holds back 20% of records as a final test set, forwarding only the remaining data to **Parameter Fitter**. We define a three-point grid for random forest depth (4, 6, 8) and a modest range for the number of trees (50, 100, 150). Orange records each trial's cross-validated metrics. Once the standard errors overlap, we freeze the configuration, export the results table, and sign the experiment log so future teammates know why the search stopped where it did.

Troubleshooting checklist

- If Orange reports zero variance across folds, confirm that shuffling is enabled; otherwise each fold might mirror the same ordering.
- When **Parameter Fitter** produces identical winners every time, audit whether the grid is too narrow to expose trade-offs.
- If the hold-out set accidentally leaks into tuning, rebuild the canvas with explicit **Data Sampler** branches to isolate the reserved data.

Computing extension. Evaluate Bayesian optimisation or adaptive grid search outside Orange, then compare their efficiency against the small fixed grids that are easiest to explain in the visual workflow.

Section summary

- Hyperparameter discipline prevents chance configurations from masquerading as genuine gains.
- Explicit audit trails—split ratios, seeds, and exported logs—preserve evaluation integrity.
- Separate validation and test pipelines shield final metrics from tuning feedback loops.

Self-check questions

1. Which signals tell you to stop exploring additional hyperparameters?
2. How does documenting split ratios help future analysts avoid data leakage?

14.3 Choose Metrics and Baselines Wisely

Summarise scoring options for classification and regression along with necessary safety checks.

Learning objectives

After reading this section, you will be able to:

- match evaluation metrics to stakeholder goals in classification and regression projects;
- compare advanced models with explicit baseline learners inside **Test & Score**;
- interpret discrepancies between metrics to trigger diagnostic investigations.

Prerequisites

Before starting, make sure you can:

- read confusion matrices and residual plots;
- describe precision, recall, and ROC AUC in plain language;
- configure baseline learners such as Orange's **Constant** model.

Cross-validation yields a table of metrics; our job is to read the whole table instead of celebrating a single column. In the Red Rooster franchise scenario we look beyond accuracy to contrast ROC AUC, F1, and Matthews correlation coefficient (MCC). Accuracy alone rewards whichever class dominates the dataset, whereas ROC AUC judges whether the model ranks actual franchise sites above the unsuitable ones, F1 balances precision and recall for class-imbalanced campaigns, and MCC condenses the confusion matrix into a correlation-style score that penalises lopsided predictions. Reading across the columns forces us to explain why a model might lift accuracy yet fall behind on recall, a conversation that surfaces stakeholder priorities before deployment.

Regression projects need equally careful scoring. We include mean squared error (MSE) and root mean squared error (RMSE) to highlight large residuals, mean absolute error (MAE) to focus on typical deviations, mean absolute percentage error (MAPE) when percentages matter, and R^2 to communicate variance explained. Whenever MAPE spikes or RMSE dwarfs MAE, we inspect the residual plots from chapter 8 to check for outliers or heteroskedasticity. Alongside these metrics we always score a simple baseline such as Orange's **Constant** model or a naive seasonal average. If our sophisticated ensemble cannot outperform the baseline across cross-validated folds, we pause and fix data quality or feature engineering issues before layering on more complexity. This guard rail keeps our evaluation honest and also prepares the separate NSW land-value regression extension later in the book.

Worked example: metric review meeting. We schedule a metrics read-out using the saved results from **Test & Score**. During the meeting, the team compares ROC AUC and F1 for each model while projecting the Orange screenshot from Figure 14.1. When the logistic regression shows higher accuracy but lower recall than the forest, we debate whether marketing cares more about not missing promising sites or about minimising unnecessary follow-ups. The discussion ends with an action item to collect additional features before promising the model to stakeholders.

Troubleshooting checklist

- If metrics fluctuate dramatically between runs, verify that class stratification is enabled during cross-validation.
- When baselines outperform complex learners, inspect feature preprocessing for leakage or mis-scaled inputs.

- If percentage-based metrics explode, remove zero-valued denominators or switch to absolute error measures.

Computing extension. Extend the evaluation table with cost-sensitive metrics tailored to your project. For example, compute expected monetary value for each threshold and compare it with ROC-derived operating points.

Section summary

- Reading multiple metrics together uncovers trade-offs hidden by accuracy alone.
- Baseline comparisons anchor whether additional modelling effort delivers real value.
- Metric anomalies prompt diagnostic dives into residuals, class balance, or feature engineering choices.

Self-check questions

1. How does MCC complement ROC AUC when judging franchise site classifiers?
2. Which metric would you prioritise when forecasting revenue, and why?

Chapter summary

- Structured cross-validation paired with sealed hold-out sets keeps reported performance trustworthy.
- Disciplined hyperparameter searches and audit trails prevent the garden of forking paths from eroding evidence.
- Metric literacy ensures we notice when sophisticated models fail to beat baselines or align with stakeholder goals.

Key term glossary

Cross-validation A resampling technique that rotates hold-out folds to estimate average model performance and its variability.

Standard error A measure of spread showing how much a metric varies across validation folds, helping us judge if differences are meaningful.

Garden of forking paths The risk that extensive, undocumented experimentation accidentally overfits evaluation data, producing misleading winners.

Baseline model A simple reference learner—such as a constant predictor—that anchors whether complex models add value.

Metric dashboard A curated table or visual summary of evaluation metrics shared during decision meetings to align stakeholders on trade-offs.

Extension challenges

- Recreate the workflows in this chapter using a different dataset with severe class imbalance. Document how stratified folds affect ROC AUC.
- Implement nested cross-validation in Python or R, then compare the variance estimates with Orange's k-fold results.
- Build a cost curve for the Red Rooster classifier and present threshold recommendations in a short executive memo.

Conclusion and next steps

Cross-validation smooths variability, disciplined searches resist the garden of forking paths, and baselines keep improvements honest. Bring these habits into Hands-on Activity 4, where we defend a classifier choice using evidence drawn from this chapter. They also carry forward into the later NSW land-value extension and the explainable-model and ensemble discussions ahead, ensuring we judge sophisticated learners by the same disciplined standards.

Chapter exercises

1. Read a short modelling workflow and mark every leakage or garden-of-forking-paths risk you can find.
2. Design a cross-validation plan for a small supervised problem. Specify folds, repeats, random seed policy, and when the hold-out set may be opened.
3. Choose a baseline for three scenarios: imbalanced classification, numeric prediction, and ranking. Explain what each baseline tests.
4. Match metrics to business questions for classification, regression, and ranking tasks. Include one case where accuracy is the wrong lead metric.
5. Fill a short lab log for a model comparison: data version, split rule, candidate models, metrics, chosen model, and caveat.

Chapter 15

Hands-on Activity 4: Compare Classifiers by Hand and in Orange

This workshop is a tactile way to compare simple classifiers without drowning in jargon. We sort coloured tokens, measure distances with rulers, mirror the same logic inside Orange, and then inspect the core hold-out idea in a short Python notebook. Every step uses plain speech while reinforcing the neighbour-based thinking from the theory chapters.

We step into this activity after exploring the neighbour tuning playbook in chapter 11, the Red Rooster classification comparison in chapter 13, and the metric guard rails in chapter 14 so the tabletop debate keeps pace with the surrounding theory chapters.

15.1 Kit Checklist

Translate the shared lab materials into a per-group kit before starting.

- **Coloured counters or tokens.** Use at least three colours so the classes are easy to distinguish.
- **Small markers for validation and test cases.** Any distinctive marker works for hiding labels until scoring time.
- **Ruler or string.** This stands in for a straight-line decision boundary during the logistic-regression part.
- **Camera or phone.** Encourage each group to photograph the layout before the tokens move.

15.2 Learning Goals

By the end of the activity, we should be able to:

- Perform a clean train/validation/test split without leaking information.

Figure 15.1: One possible classifier-comparison kit: counters, a dinosaur marker, a ruler, and thread ready for train/validation/test play on the tabletop layout.

- Compare a linear decision boundary (logistic regression—a straight-line decision boundary) against non-parametric k-NN with $k = 1$ and $k = 3$.
- Compute accuracy, precision, recall, and F1 from confusion matrices, remembering that precision answers "of the positive calls we made, how many were right?" while recall asks "of the real positives, how many did we catch?".
- Explain the bias–variance trade-off we observe between 1-NN and 3-NN.
- Rebuild the workflow in Orange, then inspect the same train/test-split idea in a short Python notebook.

15.3 Part A: Classification with Coloured Tokens

15.3.1 Setup (5 minutes)

A1. Pick two or three colours to represent classes. One student scatters all the red tokens, another the greens, another the blues, so each class has its own loose cluster with some overlap. Take a quick photo.

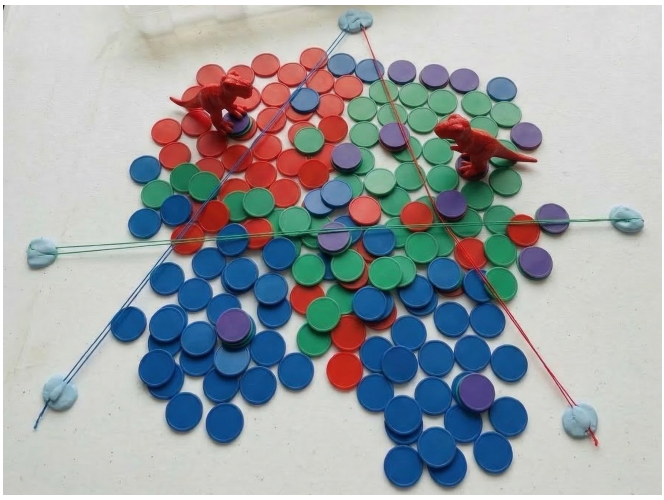
A2. However many dinosaurs you have, that is how many test instances you have. Put a dinosaur at random on some tokens and try to cover the colour. Make another 5 tokens validation instances. Come up with a scheme to identify them uniquely, for example “the validation token at the top right”. Place **purple** counters on those tokens so the original colour is hidden. Everything else is training data.

One possible end state for A2: dinosaurs mark the test tokens, purple tokens mark the validation tokens, and everything else is training data.

A3. Optional stress test: split the blue tokens into two separated piles to create a non-linear class.

15.3.2 Logistic Regression by Ruler or Thread (10 minutes)

A4. Using only the **training** tokens, place a ruler or a stretched piece of thread to form a linear decision boundary for **red vs. not-red**. If you have spare rulers, set up additional one-vs-rest boundaries.



One possible setup after A4: thread marks linear decision boundaries on the training layout. This is a pretty good boundary for blue, but maybe a bit cautious for red and green.

A5. Validate: for each **validation** token (hidden by purple), predict the class based on which side of the relevant ruler/thread boundary it sits. Record the predictions without lifting the purple tokens.

A6. Reveal the validation labels, populate a confusion matrix, and compute per-class precision, recall, F1, plus overall accuracy. Capture the layout and matrix before moving the tokens.

Remind everyone that logistic regression is simply a way to draw a straight divider and then translate distance from that divider into a probability between 0 and 1. The ruler stands in for the equation that Orange will later learn automatically.

15.3.3 k-NN on the Same Validation Set (15 minutes)

A7. Remove the ruler/thread boundaries. With the **training** tokens, run 1-NN: for each validation token, find the single nearest training token (use a ruler or a piece of thread as the measuring guide if needed) and predict its colour. Record predictions, reveal the labels, then score.

A8. Repeat with 3-NN: take the three closest training tokens and predict the majority colour (break ties consistently, for example with a coin flip or the nearest of the tied colours).

A9. Model selection: compare validation metrics for the ruler-based logistic boundary, 1-NN, and 3-NN. Declare the winner or note if two models are effectively tied.

15.3.4 Single Test Pass (5 minutes)

A10. Use the chosen model to classify the **test** tokens. Lift the dinosaur only after committing to predictions, then compute the test confusion matrix and metrics.

A11. Emphasise that adjusting decisions after seeing test labels constitutes data leakage; freeze the model once the test run begins.

15.3.5 Metric Reference

For each class c in a one-vs-rest breakdown:

$$\text{Accuracy} = \frac{TP + TN}{N}.$$

$$\text{Precision}_c = \frac{TP_c}{TP_c + FP_c}.$$

$$\text{Recall}_c = \frac{TP_c}{TP_c + FN_c}.$$

$$\text{F1}_c = \frac{2\,\text{Precision}_c\,\text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}.$$

Report macro-F1 (the mean of per-class F1) or list the class-level values directly.

Common gotchas

- If blue splits into two piles, a straight boundary struggles; expect 3-NN to outperform the ruler.
- Strong class imbalance can make raw accuracy misleading, so we focus on recall and F1 instead.
- Agree on a deterministic tie-breaker for 3-NN before scoring to keep results reproducible.

15.4 Part B: Rebuild the Workflow in Orange

15.4.1 Painted Classification (required, around 25 minutes)

Part A used an explicit train/validation/test split because we were selecting models by hand. In Orange, **Test & Score** can automate that comparison across repeated validation folds, so the software section switches to cross-validation.

C1. Open Orange and drag `Paint Data`. Recreate the Part A layout, assigning a discrete class colour to each cluster.

C2. Add `Select Columns` to confirm the painted class is the **Target**.

C3. Add two `kNN` learners (set $k \in \{1, 3\}$) and a `Logistic Regression` learner.

C4. Feed the painted data and learners into `Test & Score`. Choose 5-fold cross-validation.

C5. Record Accuracy, macro-F1, AUC, and MCC for each learner. Open `Confusion Matrix` to inspect per-class behaviour and `Scatter Plot` to compare decision boundaries.

C6. If time permits, open `ROC Analysis` from `Test & Score` so you can connect the AUC number to the curve itself.

C7. Optional: duplicate the `kNN` learner for $k \in \{1, 3, 5, 7\}$ and compare the learners side by side in `Test & Score`. This mirrors the manual validation you just performed without hiding the choices inside another widget.

15.5 Part C: Short Python Notebook

The notebook keeps the Python workload deliberately small. It uses the Telco churn data, but only asks us to load a CSV into pandas, split a table into train and test parts, and inspect how `random_state` and `test_size` change the result. Open it either in Jupyter Notebook on your own machine or in Google Colab.

Source note. The Telco churn CSV supplied for this activity is a course copy of a widely used customer-churn sample dataset. In this textbook it is used only for train/test-split practice rather than for a full churn-modelling case study.

P1. Open the supplied train/test-split notebook and complete the blank cells as you go.

P2. Load the Telco churn CSV into a pandas DataFrame and inspect a few columns.

P3. Run `train_test_split(df)` twice. Explain why the two train tables differ.

P4. Add `random_state=42` and rerun the split. What stays the same now?

P5. Add `test_size=0.2`. What fraction of the rows now lands in the test table?

P6. Identify the column that would later become the target for a churn classifier.

Debrief Prompts

- When blue was split into two piles, how did logistic regression compare with k-NN? Why?
- Did 1-NN overreact to local clutter? Did 3-NN smooth it out?
- In the notebook, why did the split change from run to run until you fixed the random seed?
- If you later wanted to build a churn classifier, which column would become the target?

We keep photos of our physical layouts alongside screenshots of the Orange workflow so we can rehearse the reasoning between sessions.

Chapter exercises

1. Record the physical classifier predictions and errors for your token layout. Identify one point each method finds difficult.
2. For two query tokens, compare nearest-neighbour reasoning with the straight-boundary reasoning used by logistic regression.
3. Recreate the layout with **Paint Data**, compare learners, and save one workflow screenshot plus one metric table.
4. In the short Python notebook, explain what `random_state` and `test_size` control. Why should test rows stay hidden while choosing the model?
5. Assemble a small evidence pack: one photo or sketch, one workflow screenshot, one metric table, and a three-sentence conclusion.

Chapter 16

Design Explainable Models with Rules and Trees

Chapter overview

We now turn evaluation discipline toward explainability. This chapter helps us decide when stakeholders need transparent models, walks through rule- and tree-based learners in Orange, and shows how to communicate their insights responsibly. The lessons lead directly into the ensemble chapter and Hands-on Activity 5, where we balance interpretability with ranking strength in a shared workspace.

16.1 Decide Whether You Need Explanations or Pure Accuracy

Start each project by clarifying whether stakeholders demand insight, automated predictions, or both.

Learning objectives

After reading this section, you will be able to:

- articulate the trade-off between interpretability and raw predictive accuracy;
- map stakeholder requirements to suitable model families in Orange;
- document decision rationales that withstand regulatory or audit scrutiny.

Prerequisites

Before starting, ensure you can:

- summarise stakeholder goals in a short design brief;

- distinguish between descriptive, predictive, and prescriptive analytics tasks;
- navigate the Orange canvas to add classification learners.

Before launching a modelling sprint we meet with stakeholders to decide whether transparent reasoning outranks raw accuracy. Some projects are classic inference workloads: we must explain why service requests spike or which regional factors influence franchise success, so the organisation can intervene. Others are industrial automation problems where millions of decisions flow through a scoring API and the only goal is a reliable yes or no. Laying out the expected workload guides our choice of modelling family and evaluation targets.

Regulated or high-stakes settings often settle the debate for us. When a system affects access, pricing, safety, or other significant outcomes, stakeholders may expect a clear explanation of why a decision was made. Even without formal legal pressure, explainable models build trust, accelerate debugging, and give product teams a more defensible story when pitching data-driven features. Documenting the rationale in a short decision memo keeps future analysts aligned with the original choice and provides evidence if auditors ask why a black-box model was rejected.

Worked example: stakeholder alignment workshop. We run a half-hour workshop before opening Orange. The operations team outlines their need to understand why certain suburbs underperform, while marketing emphasises turnaround speed for campaign approvals. Together we score priorities on a whiteboard and conclude that rule-based explanations must accompany any predictions. We then record this agreement in the project journal and tag our Orange workflow with notes explaining why CN2 and pruned trees will anchor the modelling phase. This explicit trace links stakeholder goals to modelling choices and saves time when executives revisit the decision weeks later.

Troubleshooting checklist

- If stakeholders cannot agree on priorities, draft separate evaluation dashboards for interpretability and accuracy so they can compare outcomes directly.
- When legal requirements feel ambiguous, consult compliance guidelines early and cite the relevant clauses in your modelling plan.
- If the team defaults to opaque models out of habit, prototype a transparent alternative in Orange to demonstrate what might be lost.

> **Computing extension.** Compare stakeholder alignment frameworks such as value-sensitive design or algorithmic impact assessments. Evaluate how each approach structures conversations about explainability obligations.

Section summary

- Stakeholder clarity determines whether we prioritise transparent reasoning or maximal accuracy.
- Regulatory contexts often mandate explainable pathways, especially when decisions affect individuals.
- Documenting the decision process provides future auditors with evidence that obligations were met.

Self-check questions

1. Which factors tip the balance in favour of interpretability during the workshop?
2. How would you document the rationale for selecting a black-box model when transparency is less critical?

16.2 Engineer Features that Support Human Reasoning

Augment core data with interpretable attributes so rule learners can mirror the factors humans care about.

Learning objectives

After reading this section, you will be able to:

- design feature engineering checklists that mirror stakeholder language;
- evaluate data quality after joins and imputations in Orange;
- annotate workflows so collaborators understand transformation choices.

Prerequisites

Before starting, make sure you:

- can calculate ratios and percentages from raw census attributes;
- know how to use Orange's **Formula** and **Select Columns** widgets;
- are comfortable sampling records to verify join accuracy.

Explainable modelling depends on sensible inputs. We begin with a brainstorming session to list the socio-economic signals that community managers already discuss—household income, vehicle access, cultural hubs, language communities—and note which public datasets contain them. From there we gather complementary tables, such as census counts or

transport accessibility scores, and calculate ratios that humans reason about, like average cars per household or proportion of young families per suburb. Joining these attributes carefully ensures every franchise location inherits the context we need to generate persuasive rules.

Each join deserves a quick quality check: we confirm row counts before and after merges, sample a few locations to compare against known facts, and record any imputed values so they are not mistaken for real observations later. Annotating the Orange canvas, just as we do in chapter 3, keeps those decisions close to the data. When the features align with stakeholder language, CN2 and decision trees can surface narratives that sound familiar rather than surprising jargon.

Worked example: constructing interpretable ratios. Starting from the Red Rooster feature table, we create a new Orange workflow that adds census-derived population density and cultural diversity scores. Using **Formula**, we derive "family ratio" as the proportion of households with children under ten and "mobility score" as the share of residents commuting by public transport. After each transformation we add a **Data Table** widget to inspect sample rows and confirm that values align with known suburbs. Once satisfied, we annotate the workflow with sticky notes documenting each feature's rationale so stakeholders can trace the logic when reviewing rules.

Troubleshooting checklist

- If joins drop records unexpectedly, double-check key formats and clean whitespace or casing issues before merging.
- When engineered ratios explode due to small denominators, add smoothing terms or cap extreme values before training.
- If collaborators struggle to follow the canvas, group widgets spatially and use colour-coded annotations to highlight data sources.

> **Computing extension.** Prototype feature importance calculations using Shapley-value approximations on top of your engineered features. Compare which attributes CN2 emphasises versus a black-box gradient boosting model.

Section summary

- Feature design anchored in stakeholder vocabulary keeps explainable models persuasive.
- Routine data quality checks prevent misinterpreting imputed or mismatched records.
- Well-annotated workflows accelerate peer review and onboarding.

Self-check questions

1. Which validation steps help you trust a newly engineered ratio?
2. How do workflow annotations support future maintenance of explainable models?

16.3 Induce Transparent Rules with CN2

Use rule learners to capture geographic or demographic heuristics in concise, auditable statements.

Learning objectives

After reading this section, you will be able to:

- configure the CN2 rule inducer in Orange and interpret its outputs;
- evaluate rule quality using weighted relative accuracy and coverage;
- collaborate with domain experts to refine or challenge induced rules.

Prerequisites

Before starting, confirm you:

- have engineered interpretable features as described earlier in the chapter;
- can export Orange widget outputs to CSV for offline review;
- understand confusion matrices and class balance considerations.

The CN2 rule inducer hunts for compact descriptions that separate our target classes. It follows a separate-and-conquer strategy: a beam search grows each candidate rule by adding conditions, ranks the contenders with a configurable search heuristic such as weighted relative accuracy (WRAcc), then locks in the best rule, removes the examples it covers, and searches again for the rest. Each clause reads like "If median household income exceeds $85,000 and vehicle access is high, then franchise potential is good," mirroring how franchising teams already argue their case.

Interpreting the CN2 output means scanning both the ranked rules and the default catch-all clause. Weighted relative accuracy shows how much better the rule performs compared with random guessing, while coverage tells us how many examples it applies to. We export the rule list as a CSV or PDF so domain experts can annotate it, circle clauses that feel counterintuitive, and trace misclassifications back to the feature thresholds. These discussions often inspire extra features or prompt us to collect new context before training the next iteration.

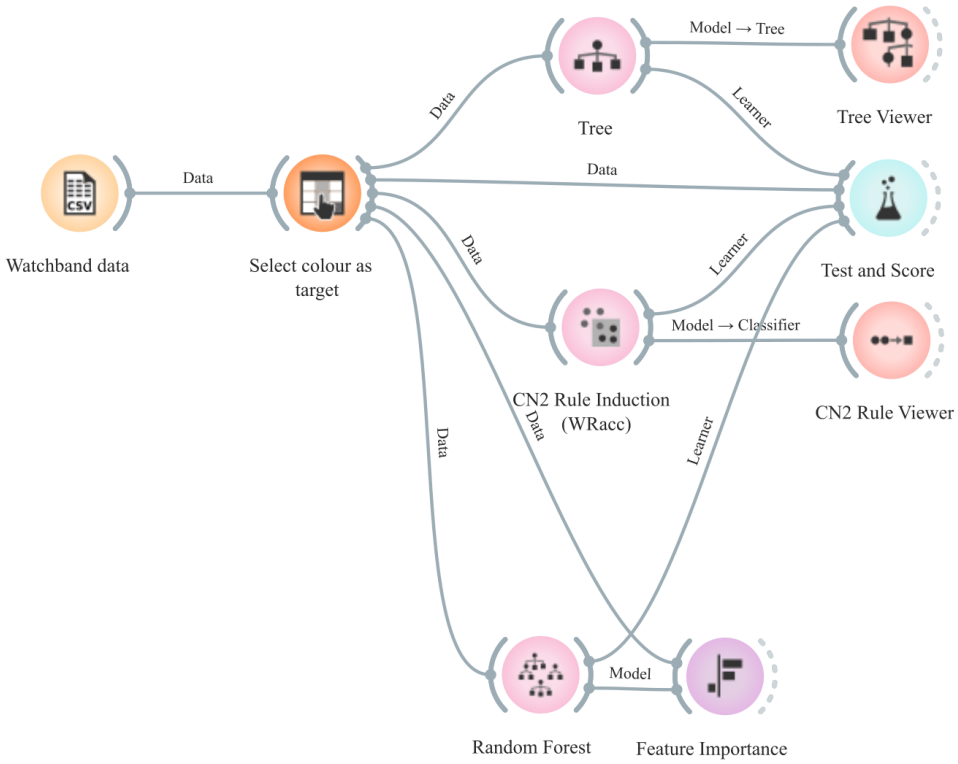


Figure 16.1: Orange workflow for the watchband explainability activity. The same labelled data feeds a decision tree, CN2 rules, a random forest baseline, shared evaluation, and feature-importance checks so students can compare transparent models against a stronger ensemble.

Worked example: co-authoring rules with experts. After training CN2 on the engineered dataset, we invite the franchise operations lead to a review session. Together we open the Orange rule viewer and sort clauses by weighted relative accuracy. When we encounter a rule stating that "high mobility score and medium income" leads to poor performance, the expert recalls a specific suburb that contradicts the clause. We flag the example, trace it back through **Data Table**, and discover that recent transport upgrades have not yet been captured in the dataset. The insight triggers a follow-up task to update mobility metrics before the next training run.

Troubleshooting checklist

- If CN2 returns overly long rules, tighten the beam width or increase the minimum covered examples to focus on meaningful segments.
- When conflicting rules appear, cluster them by shared predicates and discuss with stakeholders which clause best reflects reality.
- If coverage is too low, revisit feature engineering to ensure relevant context is available to the learner.

> **Computing extension.** Experiment with Laplace correction or entropy-based pruning parameters in CN2. Compare how these adjustments affect rule generality and stakeholder trust.

Section summary

- CN2 surfaces human-readable clauses that align with operational heuristics.
- Collaborative review ensures rules reflect up-to-date context and domain knowledge.
- Tuning rule complexity keeps explanations concise and actionable.

Self-check questions

1. How do weighted relative accuracy and coverage complement one another when ranking rules?
2. Which collaboration practices keep CN2 rules relevant as the environment changes?

16.4 Explain Decisions with Trees

Leverage decision trees when you need higher accuracy while keeping reasoning pathways accessible.

Learning objectives

After reading this section, you will be able to:

- configure tree depth, leaf sizes, and pruning options in Orange;
- narrate decision paths from root to leaf using stakeholder vocabulary;
- pair tree visualisations with evaluation metrics to justify adoption.

Prerequisites

Before starting, make sure you:

- can interpret impurity measures such as Gini and entropy;
- know how to export tree diagrams from Orange for presentation;
- understand the ensemble context provided in chapter 18.

Decision trees extend the rule-based mindset while nudging accuracy upward. Each split maximises impurity reduction—measured by Gini or entropy—so the tree quickly isolates the most informative features. We control the bias–variance trade-off by tuning maximum depth, minimum samples per leaf, or post-pruning settings: shallow trees provide high-level policies, while deeper trees trace nuanced interactions between demographics and location characteristics.

Tree visualisations show how feature space partitions into rectangles. Moving from root to leaf, we can narrate decisions as "Start with housing density, then branch on household income, and finally check commuter ratios." When we prune the tree to a few layers and export it as a PDF, stakeholders can literally print the reasoning and keep it next to their site selection checklist. This blend of clarity and modest accuracy upgrades makes trees a natural bridge to the ensemble methods we explore in chapter 18 and the Orange workflows in Hands-on Activity 5.

Worked example: pruning for presentation. We train a decision tree on the same engineered features used for CN2, setting maximum depth to six and minimum leaf size to 20. Orange's tree viewer initially displays a complex structure, so we enable post-pruning until validation accuracy plateaus while the number of leaves drops. The resulting tree contains four levels, each tied to stakeholder-friendly splits such as income bands and mobility scores. We export the diagram, highlight the key paths in a slide deck, and rehearse how to explain each branch during the next steering committee meeting.

Troubleshooting checklist

- If the tree overfits, enable pruning or reduce maximum depth to prevent brittle leaf nodes.

- When splits rely on obscure features, revisit engineering choices or adjust feature selection to emphasise interpretable attributes.
- If visualisations become cluttered, export the tree and annotate only the most important branches for discussion.

Computing extension. Analyse how surrogate splits or rule extraction techniques approximate deeper tree logic. Compare their fidelity to the original model when presenting to expert audiences.

Section summary

- Decision trees offer a balance between interpretability and accuracy when tuned thoughtfully.
- Pruning and feature selection keep tree narratives focused on stakeholder-relevant signals.
- Clear tree visualisations help us explain model logic to decision-makers.

Self-check questions

1. How does pruning influence the balance between accuracy and interpretability?
2. Which features would you highlight when narrating a tree to decision-makers?

Chapter summary

- Stakeholder alignment determines when explainability outweighs marginal accuracy gains.
- Human-centred feature engineering primes CN2 and trees to produce persuasive narratives.
- Collaborative review and pruning practices keep rule- and tree-based explanations trustworthy over time.

Key term glossary

Explainability brief A documented agreement capturing stakeholder priorities, regulatory obligations, and desired transparency outcomes.

Weighted relative accuracy CN2's improvement score over random guessing, balancing precision with rule coverage.

Coverage The proportion of instances a rule or tree path applies to, guiding whether an explanation is broadly useful.

Post-pruning A technique that removes branches from a trained tree to improve generalisation and simplify explanations.

Workflow annotation Notes embedded within the Orange canvas to record data sources, feature logic, and modelling assumptions.

Extension challenges

- Recreate the feature engineering workflow for a healthcare or energy dataset and compare how domain context shifts the induced rules.
- Translate a pruned decision tree into a natural-language policy manual for franchise selection and gather stakeholder feedback.
- Prototype a hybrid system where a black-box model triggers alerts that must be explained using CN2-derived surrogate rules.

Conclusion and next steps

This chapter paired stakeholder requirements with explainable models, interpreted rule and tree outputs, and reported them with care. Carry these habits into Hands-on Activity 5, where the watchband scenario asks us to balance transparency with performance. The next chapter on ensembles will stretch the same judgement: we will chase ranking gains without losing sight of the explanations your stakeholders now expect.

Chapter exercises

1. Audit five candidate features for explainability. Which would a stakeholder understand directly, and which need transformation or documentation?
2. Interpret a small CN2 rule list. For one rule, explain coverage, class distribution, and WRAcc in plain language.
3. Trace one decision-tree prediction from root to leaf and rewrite the path as a sentence a domain expert could challenge.
4. Compare a transparent model with a stronger ensemble. When would you recommend the transparent model even if its metric is slightly lower?
5. Draft a note inviting a domain expert to review two rules. Include what feedback would cause you to revise the data or features.

Chapter 17

Narrative Learning: Models as Instructions

Chapter overview

Narrative learning treats a readable set of instructions as the model itself. Instead of training a numerical model and then asking for an explanation afterwards, we learn a narrative that a person or a language model can apply directly to new cases. This chapter places that idea beside the rule and tree learners from chapter 16: the goal is still transparent classification, but the representation is ordinary language rather than a fitted coefficient table, tree diagram, or Orange rule list. The final section turns the method into a low-friction practical that can run with paper cards, a spreadsheet, or a carefully logged language-model workflow.

17.1 Treat the Explanation as the Model

Use a human-readable rulebook as the object being learned, not merely as commentary on a hidden model.

Learning objectives

After reading this section, we will be able to:

- distinguish narrative learning from post-hoc model explanation;
- describe the author–executor loop that turns examples into revised instructions;
- explain why narrative models belong in the same conversation as rules, trees, and other explainable learners.

Prerequisites

Before starting, make sure you can:

- read a confusion matrix and identify false positives and false negatives;
- describe how rule learners and decision trees produce transparent decision logic;
- explain why a held-out test set must not influence model design.

Most explainability workflows keep two objects. First, there is the model that makes the prediction. Second, there is an explanation that tries to describe what the model did. A decision tree narrows that gap because the model is already close to a readable flow chart. A logistic-regression coefficient table is less direct, but still inspectable. A large black-box model sits further away: we may add feature attributions, local surrogate rules, or example-based explanations, but those artefacts are not the decision process itself.

Narrative learning pushes the idea in the other direction. The readable explanation is the model. A narrative might say, "If the applicant has repeated late payments and the income band is low, classify the risk as high unless there is a recent repayment record." The classifier then applies those words directly to each row. If the narrative is incomplete, inconsistent, or too vague, we treat that as a modelling failure rather than a communication failure.

That framing is useful because it makes the model auditable at the point of use. A stakeholder can read the same instructions that the executor applies. A tutor can ask a student to classify one case by hand and compare their result with the automated executor. The price is that the narrative must be written with unusual care. Ambiguous phrases such as "fairly high" or "usually suspicious" produce inconsistent inference unless the surrounding context pins them down.

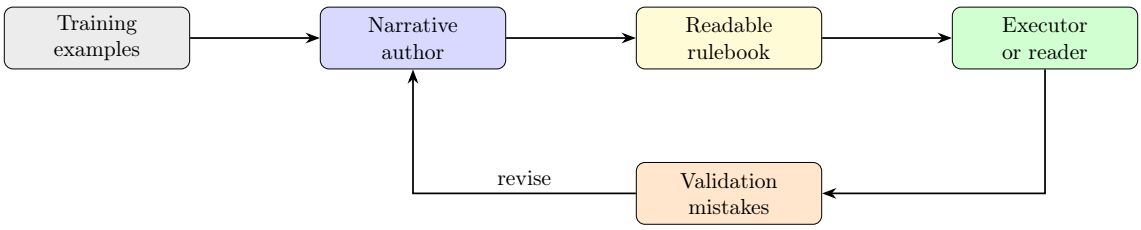


Figure 17.1: Narrative learning loop. A narrative author writes instructions, an executor applies them to cases, and validation mistakes drive the next revision.

The research note that motivates this chapter used the names "overseer" and "underling" for these two roles [13]. For teaching, the plainer names *author* and *executor* are easier. The author proposes the rulebook. The executor follows it. The validation set reports where the rulebook fails. The author then revises the rulebook and tries again.

Section summary

- Narrative learning makes the explanation the executable model.
- Ambiguous language becomes a technical defect because the executor must apply the narrative case by case.

- The author–executor loop gives us a practical way to improve the narrative using validation mistakes.

Self-check questions

1. Why is a narrative model different from a post-hoc explanation of a random forest?
2. What kinds of wording would make a narrative difficult for two readers to apply consistently?

17.2 Train a Narrative with Examples and Mistakes

Iterate from a weak rulebook toward clearer instructions by studying where the current narrative fails.

Learning objectives

After this section, we will be able to:

- run a simple narrative-learning cycle with training, validation, and test splits;
- organise false positives and false negatives into useful revision evidence;
- decide when to stop revising a narrative before it overfits the validation examples.

Prerequisites

Before proceeding, confirm that you can:

- split a labelled dataset into training, validation, and test portions;
- calculate accuracy, precision, recall, and F1 score from predicted labels;
- explain why changing the model after viewing test errors invalidates the test.

A narrative-learning workflow begins like any other supervised classification project. We start with labelled examples and hold back a test set. The training examples are the only examples the author may use when writing rules. The validation examples measure whether the latest narrative is improving. The test examples stay sealed until the end.

The first narrative can be deliberately weak. It might say "choose randomly" or "predict the majority class." The point is not to be clever at round one. The point is to generate mistakes that reveal what the next rulebook should address. After the executor applies the current narrative, we sort validation rows into true positives, false positives, true negatives, and false negatives. A small balanced sample from those groups gives the author concrete evidence: "these cases were missed; these cases were incorrectly included; these cases worked."

A basic training cycle.

Step 1: Split the labelled data into training, validation, and test sets.

Step 2: Ask the author to write an initial narrative using only the training examples and the target definition.

Step 3: Give the narrative and one row at a time to the executor. Record the predicted label for each validation row.

Step 4: Summarise the mistakes: false positives, false negatives, and a few correctly classified examples.

Step 5: Ask the author to revise the narrative so that at least one observed mistake would now be handled correctly.

Step 6: Repeat until the validation score stops improving for a fixed number of rounds.

Step 7: Freeze the final narrative, then evaluate it once on the held-out test set.

This loop looks informal, but it still follows ordinary model-selection discipline. The author is tuning the narrative against validation evidence. If the author sees the test errors and keeps editing, the test set becomes another training source. That is no better than tuning a random forest on the test set and then pretending the final score is independent.

Worked example: revising a policy narrative. Suppose the task is to classify whether a support ticket needs urgent escalation. The first narrative says, "Escalate billing problems and security problems." The executor then misses several rows where the customer threatens to cancel, and incorrectly escalates routine invoice-copy requests. The next narrative becomes more precise: "Escalate security problems, payment failures that block account access, and tickets that explicitly mention cancellation or legal action. Do not escalate routine requests for invoices, receipts, or account details unless another escalation condition appears." The revision is not just longer. It repairs a specific false-negative pattern and a specific false-positive pattern.

Three habits keep the cycle productive. First, each revision should add or refine rules rather than merely describe the desired behaviour. "Be more careful with cancellation" is not a narrative model; "if the text contains a cancellation threat, escalate unless it is only asking how to cancel a trial" is closer. Second, revisions should be logged so we can reconstruct how the final narrative emerged. Third, the stopping rule should be decided before the author gets tired or delighted. A common choice is a small patience value: stop after several rounds with no validation improvement.

Section summary

- Narrative learning uses validation mistakes as structured feedback for rulebook revision.

- False positives and false negatives are more useful than a single accuracy number because they show what the narrative needs to repair.
- A frozen final narrative must be tested only once on the held-out test set.

Self-check questions

1. Why is a sample of false negatives often more useful than a single validation score?
2. What would make a narrative revision an overfit response to the validation set?

17.3 Evaluate Narratives as Models

Judge narratives by predictive performance, readability, stability, and auditability.

Learning objectives

After this section, we will be able to:

- evaluate a narrative model with the same held-out discipline used for other classifiers;
- identify failure modes that are specific to language-based inference;
- explain why readability is an evaluation criterion, not decorative polish.

Prerequisites

Before starting, ensure you can:

- compare models against a simple baseline;
- read a short rule list and trace its decision path for one example;
- describe how random variation can affect repeated model runs.

Narrative models should never receive a free pass because they are readable. We still compare them against baselines: majority-class prediction, logistic regression, a shallow decision tree, CN2 rules, or whichever model family suits the dataset. The narrative earns its place only if its combination of performance and interpretability justifies the extra labour.

At the same time, narratives have evaluation dimensions that ordinary classifiers do not expose as clearly. The most obvious is readability. If the final rulebook is so tangled that a trained reader cannot apply it reliably, then its claim to explainability is weak. We can test this by asking two students or auditors to classify the same small sample from the frozen narrative and measuring agreement. Disagreements reveal ambiguous thresholds, missing tie-breakers, or overloaded language.

Stability also matters. If the author is a language model, repeated runs may produce different narratives. If the executor is a language model, the same narrative may be applied inconsistently unless the task is tightly specified. Frontier models usually follow explicit instructions more reliably than weaker models, but we still log model names, prompts, temperatures, and output formats. If humans act as executors, we record their questions and disagreements because those are genuine defects in the narrative.

Common failure modes.

- **Vague thresholds:** The narrative says a value is "large" or "recent" without defining the boundary.
- **Contradictory clauses:** Two rules apply to the same case but point to different labels.
- **Hidden prior knowledge:** The author recognises a famous dataset and imports outside facts instead of learning from the examples.
- **Executor drift:** The executor paraphrases or "improves" the rulebook instead of following it.
- **Validation memorisation:** Later narratives patch individual examples without learning a general rule.

The hidden-prior-knowledge problem deserves special care in teaching. If we use the Titanic dataset unchanged, many people and many language models already know that sex and passenger class are strong predictors of survival. A narrative that rediscovers this pattern may be drawing on background knowledge rather than the provided examples. One research response is to transform a known dataset into a disguised story while preserving the feature relationships. For a classroom exercise, a fresh synthetic dataset or a hand-made card deck is usually simpler.

Section summary

- Narrative models should be evaluated against ordinary baselines, not treated as automatically better because they are readable.
- Readability, inter-reader agreement, and inference stability are part of the model-quality check.
- Disguised or synthetic datasets reduce the risk that the author solves the task from prior knowledge.

Self-check questions

1. How would you test whether two people interpret a narrative rulebook in the same way?
2. Why might a famous dataset make a narrative-learning experiment misleading?

17.4 Classroom Practical: Learn a Rulebook

Run narrative learning as a physical or spreadsheet activity before automating any part of it.

Learning objectives

After this section, we will be able to:

- organise a narrative-learning practical with clear student roles;
- collect enough evidence to compare narrative rules with a baseline classifier;
- adapt the exercise for paper cards, spreadsheets, or language-model executors.

Prerequisites

Before running the practical, students should be able to:

- calculate a small confusion matrix by hand;
- distinguish training, validation, and test rows;
- write unambiguous if–then rules.

This practical is deliberately easy to organise. Each group receives a small labelled dataset printed as cards or rows in a spreadsheet. The task might be classifying support tickets, deciding whether a fictional product should be recalled, or identifying whether a transformed scientific observation belongs to class A or class B. The important requirement is that the features are concrete enough to support rules, but not so obvious that the first narrative is perfect.

Role	Sees labels?	Main job	Evidence produced
Narrative author	Training only	Write and revise rules	Versioned rulebook
Executor	No	Apply rules to rows	Predicted labels
Auditor	Yes, after prediction	Build confusion matrix	Error summary
Domain challenger	Training only	Ask whether rules make sense	Ambiguity notes

Table 17.1: Suggested student roles for a paper-based narrative-learning practical.

The first round takes about ten minutes. The author studies the training rows and writes a one-page rulebook. The executor applies it to the validation rows without seeing

labels. The auditor reveals the validation labels, builds the confusion matrix, and prepares a short error pack: two false positives, two false negatives, and two correct examples if available. The author revises the rulebook and the group repeats the cycle for two or three rounds.

The final round uses the sealed test rows. At that point the author may not revise. Groups report the final accuracy and one example where their narrative feels genuinely explanatory. They also report one ambiguous case where two reasonable readers might disagree. That last discussion is the heart of the practical: an explainable model is not just a model with words attached. It is a model whose words can survive another person's careful reading.

Optional language-model variant. If the class has approved access to a language model, the same structure can be automated. The narrative author becomes a prompt that asks for explicit rules. The executor becomes a separate prompt that receives the frozen rulebook and one row at a time. Keep the two prompts separate, log every version, and use fictional or sanitised data. Do not use personal, medical, financial, or assessment data for a convenience demo.

Comparison extension. After the test round, students can fit a shallow decision tree or rule learner to the same dataset. The comparison question is not simply "Which score is higher?" Ask which model a stakeholder could apply, audit, or challenge more easily. A narrative that performs slightly worse may still be preferable when the rulebook captures the decision policy clearly. A narrative that performs slightly better may still be rejected if it relies on muddled language that readers cannot apply consistently.

Chapter summary

- Narrative learning makes the natural-language rulebook the model itself.
- The training loop improves the rulebook by feeding validation mistakes back to the narrative author.
- Evaluation must include predictive metrics, baseline comparisons, readability, stability, and auditability.
- The practical can run with paper cards and student roles before any LLM automation is introduced.

Key term glossary

Narrative model A human-readable set of instructions that directly produces predictions.

Narrative author The person or model that proposes and revises the rulebook.

Executor The person or model that applies a fixed rulebook to one case at a time.

Error pack A small structured sample of false positives, false negatives, and correct classifications used to guide revisions.

Executor drift A failure mode where the executor changes, expands, or interprets the rulebook instead of following it.

Extension challenges

- Run the same dataset with human authors and language-model authors, then compare rule length, accuracy, and inter-reader agreement.
- Test whether adding more validation examples per round improves the final narrative or merely overwhelms the author.
- Convert a final narrative into a decision tree and identify where the tree loses nuance or the narrative loses precision.

Conclusion and next steps

Narrative learning belongs naturally after rules and trees because it asks the same question in a sharper form: can the decision logic itself be the artefact we give to people? The answer depends on disciplined validation, careful language, and honest auditing. The next ensemble and evaluation chapters keep the same habit of comparing transparency with performance rather than assuming either one wins automatically.

Chapter exercises

1. Convert a small set of labelled examples into an initial natural-language rulebook. Keep the rules short enough that another group can apply them.
2. Revise the rulebook after seeing validation mistakes. Mark which rule changed and what evidence forced the change.
3. Apply the final rulebook to unseen cases without editing it. Record errors separately from proposed improvements.
4. Acting as an auditor, identify one ambiguous phrase, one hidden assumption, and one likely overfit rule.
5. Write a governance note describing what evidence should be saved if the narrative is used in a real workflow.

Chapter 18

Rank Cases and Combine Models with Ensembles

Chapter overview

We close this sequence by pairing explainability with ranking power. The chapter unpacks ROC curves as ranking tools, explores when to reach for ensembles, and shows how to communicate their benefits without overstating certainty. Each section builds on the evaluation guard rails and prepares us for Hands-on Activity 5, where we weigh tree-based transparency against ensemble strength on the watchband dataset.

18.1 Read ROC Curves as Ranking Tools

Treat AUC as evidence that your model orders cases well, not as proof that its probabilities are trustworthy.

Learning objectives

After reading this section, you will be able to:

- interpret ROC curves produced by Orange's **Test & Score** widget;
- explain why AUC reflects ranking quality rather than probability calibration;
- link ROC analysis to the broader evaluation safeguards from chapter 14.

Prerequisites

Before starting, ensure you:

- understand true positive and false positive rates;
- can configure Orange learners and evaluation widgets to output ROC curves;

- have baseline familiarity with precision–recall trade-offs.

Receiver operating characteristic (ROC) curves plot the true positive rate against the false positive rate as we slide a decision threshold from lenient to strict. Each point on the curve shows how much positive coverage we gain for the false alarms we are willing to tolerate. The area under the curve (AUC) summarises that ranking behaviour: it is the probability that a randomly chosen positive example receives a higher score than a randomly chosen negative one. AUC therefore tells us how confidently the model sorts cases, not whether the absolute probability values are calibrated.

We always pair ROC analysis with sanity checks. Accuracy and baseline comparisons from chapter 14 confirm that a higher AUC still translates into practical gains, and calibration plots reveal whether the scores line up with observed frequencies. When two models have similar AUCs, we examine the steepness of the initial curve segment: a sharp early rise means we can capture most positives with few false alarms, which is invaluable for triaging limited human attention.

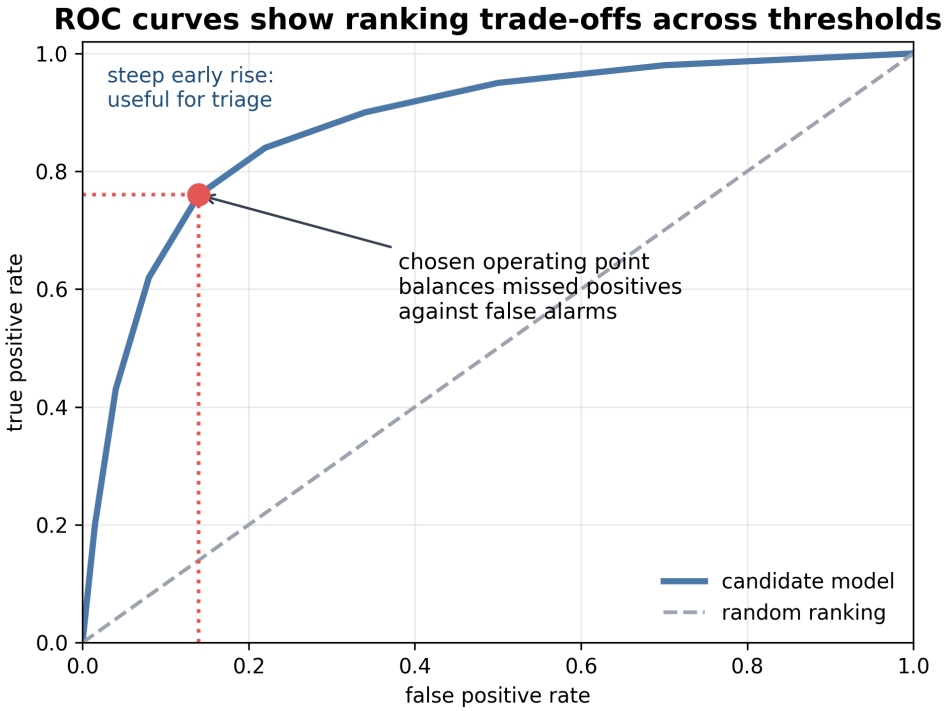


Figure 18.1: ROC curves summarise ranking behaviour across thresholds. Choosing an operating point is a separate decision that should reflect the cost of missed positives and false alarms.

Worked example: interpreting ROC in Orange. Using the workflow from Figure 14.1, we open the ROC visualiser attached to **Test & Score**. The random forest and logistic regression curves appear on the same axes, letting us observe how quickly each model lifts true positives. The forest's curve hugs the top-left corner, indicating strong ranking ability, while the baseline lags along the diagonal. We export the chart, annotate the operating

points under consideration, and paste the image into the evaluation report for stakeholders.

Troubleshooting checklist

- If ROC curves look identical across models, verify that class shuffling and stratification are active to avoid fold artefacts.
- When AUC appears high but precision remains low, inspect calibration plots and confusion matrices to detect poorly chosen thresholds.
- If Orange cannot generate ROC curves, ensure the learners output class probabilities rather than hard labels.

Computing extension. Compare ROC analysis with precision–recall curves on imbalanced datasets. Quantify when one metric offers clearer guidance than the other.

Section summary

- ROC curves reveal how models trade true positives for false positives as thresholds vary.
- AUC measures ranking ability but must be paired with calibration and baseline comparisons.
- Visual overlays help teams debate which model best supports limited operational capacity.

Self-check questions

1. Why does AUC capture ranking quality rather than probability accuracy?
2. When would you prefer a precision–recall curve over ROC?

18.2 Set Thresholds to Match Business Costs

Move decision cut-offs based on the relative harm of false positives and false negatives instead of defaulting to 0.5.

Learning objectives

After reading this section, you will be able to:

- translate stakeholder cost discussions into decision thresholds;
- use Orange outputs to evaluate alternative operating points;
- document threshold rationales for future audits.

Prerequisites

Before starting, make sure you:

- can interpret confusion matrices and cost tables;
- know how to export evaluation results from Orange for offline analysis;
- understand the business context driving classification decisions.

Ranking metrics are only the beginning—we still need to pick a threshold that reflects the stakes. We map out the cost of a false positive versus a false negative with stakeholders, then use ROC or cost curves to find the operating point that minimises expected loss. Sometimes we end up far from 0.5: a fraud detection system might trigger an investigation at 0.2 if missed cases are expensive, while a marketing campaign might require 0.7 to avoid spamming uninterested customers.

Once we select the threshold, we translate it into operational guidance. For probability-based models we note the exact cut-off; for ensembles that vote, we specify how many base learners must agree before we accept a positive. We capture the rationale alongside the metric dashboard so future evaluations respect the original trade-off, especially when new data drifts or leadership changes. In Orange, ROC Analysis gives the ranking context. If we need a formal cost comparison across several thresholds, it is better to export the predicted probabilities and compute that comparison explicitly than to imply a single widget has already solved the business decision for us.

Worked example: choosing a franchise approval threshold. We meet with the franchise steering group to quantify costs: approving a poor site wastes $250,000 in fit-out expenses, while delaying a strong site costs $80,000 in lost revenue. We use Orange's ROC view to compare ranking quality, then export the predicted probabilities and evaluate a small set of candidate thresholds in a separate table. A cut-off of 0.63 keeps the false positive rate low while maintaining recall above 0.75 for this case. We document the calculation, save the chosen operating rule in the workflow notes, and record the decision in the project log.

Troubleshooting checklist

- If stakeholders struggle to articulate costs, facilitate a scenario-planning exercise that estimates worst-case impacts.
- When cost curves are unavailable, approximate expected value manually using exported confusion matrices.
- If performance drifts after deployment, schedule quarterly reviews to revisit the threshold with fresh evidence.

Computing extension. Implement adaptive thresholding that updates based on rolling cost estimates. Compare its performance against a fixed threshold in a backtesting experiment.

Section summary

- Thresholds should encode the real-world costs of false positives and negatives.
- Documented rationales keep future audits aligned with the original decision context.
- Regular reviews prevent drift from eroding carefully chosen operating points.

Self-check questions

1. How would you explain the trade-off between recall and false positives to a non-technical stakeholder?
2. Which signals indicate it is time to revisit an operational threshold?

18.3 Stack Diverse Learners for Complementary Strengths

Combine rules, trees, neighbours, and baselines when each captures different patterns in the data.

Learning objectives

After reading this section, you will be able to:

- assemble ensembles in Orange's **Stacking** widget with diverse base learners;
- evaluate ensemble performance against individual models using cross-validation;
- maintain experiment logs that clarify why each learner was included.

Prerequisites

Before starting, ensure you:

- understand the strengths and weaknesses of logistic regression, k-NN, CN2, and random forests;
- can configure preprocessing steps such as scaling or feature selection;
- know how to interpret Orange's aggregated evaluation tables.

Stacking allows us to combine multiple modelling perspectives. We start with a diverse bench: logistic regression for linear structure, k-NN for local geometry, CN2 or trees for interpretable rules, and a simple baseline. Orange's **Stacking** widget fits a meta-learner on top of those base learners so the final prediction can learn when each model tends to be most helpful.

Diversity matters more than quantity. Adding near-duplicate models introduces noise without real signal gain, so we curate the ensemble deliberately. We log which base learners entered the stack, the validation scores that justified them, and any preprocessing steps (such as feature scaling) they required. This documentation keeps the ensemble maintainable and sets the stage for automating the workflow when the project outgrows manual Orange canvases.

Worked example: documenting a stacked ensemble. We extend the workflow from Figure 14.1 by introducing Orange's **Stacking** widget. Logistic regression, k-NN with normalised features, CN2, and random forest feed into the ensemble. After running cross-validation, we record that the stacked model lifts ROC AUC by 0.03 compared with the best individual model. We capture these results in a shared spreadsheet, noting preprocessing requirements and learner settings so future teammates can reproduce the stack.

Troubleshooting checklist

- If the ensemble underperforms, audit whether base learners are too similar or whether preprocessing created leakage.
- When Orange reports inconsistent improvements, check that each learner receives identical training folds.
- If collaboration becomes difficult, snapshot the workflow with versioned filenames and link them in the experiment log.

Computing extension. Compare Orange's stacking workflow with a soft-voting or custom meta-learner implementation in Python. Analyse when each approach offers the best balance of accuracy and interpretability.

Section summary

- Diverse base learners capture complementary patterns and lift ensemble performance.
- Careful documentation of preprocessing and validation settings keeps ensembles reproducible.
- Stacked ensembles provide a practical way to combine complementary learners without pretending every model contributes equally.

Self-check questions

1. Which criteria do you use to decide whether a learner deserves a place in the ensemble?
2. How would you document ensemble settings so a teammate can rebuild the workflow?

18.4 Adopt Random Forests and Lightweight Tuning

Lean on tree ensembles as reliable baselines while tuning key hyperparameters with cross-validation.

Learning objectives

After reading this section, you will be able to:

- explain how random forests reduce variance through bootstrapping and feature subsampling;
- tune key hyperparameters using Orange's **Parameter Fitter**;
- communicate feature importance insights to stakeholders.

Prerequisites

Before starting, make sure you:

- understand decision tree mechanics and impurity measures;
- can interpret cross-validated metric tables from **Test & Score**;
- know how to export feature importance plots from Orange.

Random forests average the predictions of many decision trees that each see different feature subsets and bootstrapped samples. Randomising the split candidates reduces correlation between trees, which lowers variance and makes forests robust on tabular data. They become our go-to baseline when we need strong accuracy without extensive feature engineering.

We still tune a few high-impact parameters. Orange's **Parameter Fitter** explores the number of trees or the maximum depth, using cross-validation from chapter 14 to judge stability. Because the widget adjusts one parameter at a time, we note any interactions to revisit later with script-based tooling. Even when individual trees are opaque, feature importance plots summarise which variables drive predictions, giving us a talking point for stakeholders who need at least directional explanations. These insights close the loop with the explainable modelling chapter while preparing us to escalate towards gradient boosting in future iterations.

Worked example: tuning a random forest baseline. We start with 100 trees and a maximum depth of eight. **Parameter Fitter** evaluates tree counts of 50, 100, and 150 across ten folds. The results show diminishing returns beyond 100 trees, while increasing depth to ten introduces higher variance. We lock in 100 trees with depth eight, export the feature importance chart highlighting income, mobility, and population density, and share the summary with stakeholders alongside CN2 and tree explanations.

Troubleshooting checklist

- If training slows dramatically, reduce the number of trees or parallelise runs outside Orange.
- When feature importance seems unstable, rerun cross-validation with different seeds to confirm the signal.
- If forests dominate every comparison, re-evaluate whether simpler, more interpretable models would suffice given stakeholder needs.

> **Computing extension.** Compare Orange's random forest with gradient boosting implementations. Analyse performance gains relative to the additional tuning burden.

Section summary

- Random forests provide strong baselines through variance reduction and feature subsampling.
- Light-touch tuning balances accuracy improvements with reproducibility.
- Feature importance charts keep ensembles connected to explainability goals.

Self-check questions

1. Which hyperparameters deliver the largest performance gains for random forests in Orange?
2. How do you explain feature importance rankings to non-technical stakeholders?

Chapter summary

- ROC literacy ensures we judge ranking quality with clear eyes before locking in thresholds.
- Cost-aware thresholding, ensemble design, and random forest tuning work together to deliver reliable classifiers.
- Documentation and stakeholder dialogue connect technical metrics to the decisions they inform.

Key term glossary

ROC curve A plot showing the trade-off between true positive and false positive rates as the decision threshold varies.

Operating point The threshold or ensemble vote that defines when a model classifies an instance as positive.

Stacking An ensemble strategy that trains a meta-learner on the outputs of multiple base learners.

Feature importance A ranking that summarises how much each input contributes to the predictions of a model such as a random forest.

Threshold audit A documented review that revisits operating cut-offs in light of new costs or observed drift.

Extension challenges

- Recompute ROC and cost analyses for a highly imbalanced dataset and compare outcomes with precision–recall curves.
- Prototype a meta-learner ensemble using Python scripts, then benchmark it against Orange's **Stacking** widget configuration.
- Conduct a quarterly threshold audit using simulated drift scenarios and summarise recommended adjustments for leadership.

Conclusion and next steps

We now know how to read ROC curves critically, tune ensembles for ranking strength, and communicate their trade-offs with stakeholders. Take these insights into Hands-on Activity 5: when your group debates whether to favour CN2 or random forests for the watchband brief, use the evidence frameworks developed here. Looking beyond the activity, keep your monitoring and calibration plans in play—they ensure that any ensemble you deploy continues to earn trust over time.

Chapter exercises

1. Given a small score-and-label table, sort cases by score and mark how the ROC curve changes as the threshold moves.
2. Use a simple false-positive and false-negative cost table to choose an operating threshold. Explain who bears each cost.
3. Compare three candidate learners and decide which one adds useful diversity to an ensemble rather than duplicating another learner's errors.
4. Interpret a small random-forest tuning table. Choose a setting that balances metric gain, variance, and complexity.
5. Explain why the model with the highest ROC AUC may still need a different threshold for deployment.

Chapter 19

Hands-on Activity 5: Trees, Rules, and Forests on the Watchband Deck

We use printed customer cards and wooden sticks to see how rule-based models make decisions before trusting the same logic inside Orange. By physically drawing lines between people, we can explain every split in plain language, then confirm the computer version matches. The goal is to keep explainable AI concrete and friendly.

Our watchband investigation assumes the transparency commitments from chapter 16 and the ranking trade-offs in chapter 18, keeping the hands-on comparisons grounded in the theory chapters that frame explainable ensembles.

This activity walks us from a tangible decision-tree simulation through CN2-style rule learning and into an Orange workflow. The physical exercise uses 18 printed customer records describing watchband purchases. Age acts as the vertical axis, income as the horizontal axis, and colour is the class label.

19.1 Kit Checklist

Allocate supplies in advance so each table can run the full sequence without waiting.

- **One printed watchband card deck per group.** Use durable cards if the activity will be reused often.
- **Simple split markers.** Wooden sticks, rulers, or strips of card all work for marking candidate splits.
- **Table space and camera.** Each group needs enough room to lay out the cards and photograph the result before moving to Orange.
- **Paper for calculations.** Each group needs somewhere to record class counts, Gini values, and CN2-style rule scores.

Appendix A.2 reproduces the A4 watchband card deck so the textbook now carries the print-ready sheets needed for the tabletop exercise.

19.2 Learning Goals

By the end of the activity, we will be able to:

- Explain how Gini impurity scores guide split selection in decision trees.
- Run a short beam-search procedure that mirrors CN2 rule induction.
- Compare physical rules and trees with the Orange implementations of tree, CN2, and random forest learners.
- Interpret feature importance scores from an ensemble and relate them back to the physical splits.

19.3 Part A: Decision Trees with Wooden Sticks

19.3.1 Lay out the dataset

A1. Arrange the 18 cards on a clear table so income increases along the horizontal axis and age increases vertically. The exact spacing is flexible; preserve the ordering rather than the scale.

A2. Keep a generous border around the grid so you can insert wooden sticks without disturbing neighbouring cards.

A3. Photograph the initial layout for later comparison.

Figure 19.1: Watchband cards arranged on a table with wooden sticks marking axes so age rises upward and income moves left to right before any splits are placed.

19.3.2 Score candidate splits

Discuss where a single vertical or horizontal wooden stick could create two purer subsets. For any proposed split:

S1. Tally the orange (r), blue (b), and green (g) cards on one side of the stick and record the total $N = r + b + g$.

S2. Compute the Gini index $G = 1 - \frac{r^2+b^2+g^2}{N^2}$ for that half.

S3. Repeat for the opposite side and add the two impurity scores, weighted by each side's size, to compare candidates.

Choose the lowest-impurity split and leave the wooden stick in place to mark the root decision.

Explain that a Gini index of 0 means the split side holds only one colour (perfectly pure). With three classes, the largest value occurs when the colours are evenly mixed, giving $G = 2/3$. So values closer to 2/3 reveal the messiest groups, and lower numbers are always better.

19.3.3 Extend the tree

T1. Divide the group so each person focuses on one branch. Repeat the scoring process within that subset to identify the best child split.

T2. Stop early if a branch is pure or contains two or fewer cards; otherwise continue until every leaf is pure or tiny.

T3. Sketch the resulting decision tree, labelling each node with its split rule and class distribution. Capture another photo before tidying up the sticks.

19.4 Part B: CN2 Rule Induction Game

CN2 learns *if–then* rules using a separate-and-conquer strategy. Recreate a simplified beam search on the card layout before turning to software.

19.4.1 Key ideas recap

- Hypotheses are conjunctions of simple literals such as "Age > 24" or "Income $\leq 90\,000$".
- The star operator specialises a rule by adding one additional literal, shrinking the covered rectangle on the grid.
- Maintain a beam of the top $B = 3$ candidate rules, refreshing it each round.
- Once a high-quality rule is accepted, remove the cards of the target class that it covers and search again for the remaining positives.

Make it clear that the "beam" is just a short list of the best candidates we carry forward each turn so the search stays manageable.

This is a teaching approximation to CN2 rather than a full implementation. The goal is to make the search strategy visible, not to reproduce every optimisation detail from the software.

19.4.2 Weighted Relative Accuracy

Score rules with Weighted Relative Accuracy (WRAcc) for a target class c:

$$\text{WRAcc} = \frac{n}{N}\left(\frac{p}{n} - \frac{P}{N}\right) = \frac{p}{N} - \frac{n}{N}\cdot\frac{P}{N},$$

where N is the dataset size, P is the number of class-c cards overall, n is the number of cards covered by the rule, and p is the number of covered cards of class c.

For example, the rule `Age` $>$ `24 & Income` $\leq$ 90 000 $\Rightarrow$ `Green` has $(p, n, P, N) = (5, 6, 6, 18)$ and a WRAcc of $\frac{1}{6} \approx 0.167$. WRAcc compares the rule's hit rate with the baseline share of the class, so positive values mean the rule finds more target cases than guessing by overall frequency alone.

19.4.3 Play one beam search per class

C1. Choose a target class (for instance, `Blue`). Record P and $N = 18$ once.

C2. Build the initial beam from all one-literal rules that mirror the valid wooden-stick positions. Keep the top three by WRAcc.

C3. Specialise each kept rule by adding one more literal that still covers at least one target-class card. Re-score, keep the top three, and repeat until WRAcc no longer improves or the rule reaches accuracy ≥ 0.8 with support $n \geq 3$. Note that these are two different criteria: WRAcc ranks the candidate rules against one another, while the accuracy threshold is the stopping rule that decides when a single rule is good enough to accept.

C4. Add the best rule to your ruleset, remove the covered target-class cards from the table, and return to step **C1** for the remaining positives. Move to the next class once all positives are covered.

Compare the final rule list against the regions defined by your physical decision tree: they should describe similar rectangles.

19.5 Part C: Rebuild the Workflow in Orange

Transition to Orange to validate the manual reasoning.

O1. Load the supplied `watchband.csv` file with a `File` widget and confirm the target is the colour column.

O2. Add `Tree`, `CN2 Rule Induction`, and `Random Forest` learners. Set the CN2 widget's evaluation measure to WRAcc to match the tabletop exercise.

O3. Connect all learners to `Test & Score` and choose 5-fold cross-validation.

O4. Inspect `Tree Viewer` to see if the learned splits replicate your wooden-stick layout, then open `CN2 Rule Viewer` to compare rules. Finally, use `Rank` or `Random Forest` feature importance to discuss why age or income dominated the decisions.

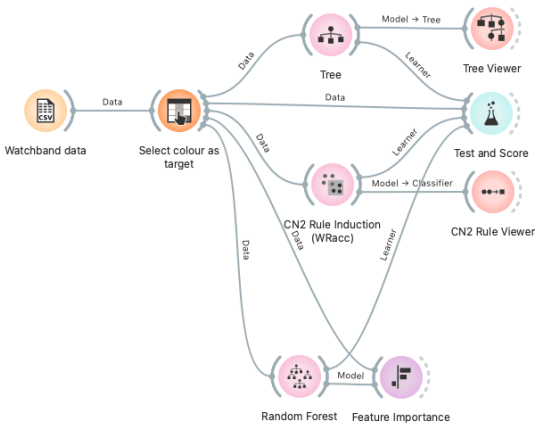


Figure 19.2: Orange workflow with File feeding Tree, CN2 Rule Induction, and Random Forest widgets into Test & Score, mirroring the physical card exercise with explainable and ensemble models.

19.6 Watchband Dataset

The practical uses the following 18-row dataset. Keep a printed copy near each table for quick reference.

#	Age	Income	Colour
1	12	0	Orange
2	14	10	Orange
3	15	0	Orange
4	17	500	Blue
5	18	10 000	Orange
6	19	30 000	Blue
7	21	0	Blue
8	21	20 000	Blue
9	23	30 000	Blue
10	25	40 000	Green
11	27	100 000	Orange
12	30	80 000	Green
13	40	150 000	Orange
14	41	60 000	Green
15	60	100 000	Green
16	62	40 000	Blue
17	63	60 000	Green
18	65	30 000	Green

Each group should archive their physical notes and Orange results so they can revisit the decision regions when studying ensemble methods.

Chapter exercises

1. Record each physical decision-tree split and the class counts on both sides. Mark the split that did the most useful work.

2. Choose one proposed CN2 rule and compute its coverage. Then explain what WRAcc is trying to reward.

3. Compare the rule list with the decision tree. Give one reason the rules are easier to explain and one reason the tree is easier to follow.

4. Load `watchband.csv` in Orange, compare tree, CN2, and random forest learners, and save a metric table.

5. Write a debrief paragraph recommending whether the watchband scenario should prioritise accuracy, explainability, or a compromise.

Part II

Mathematics, Programming, and Extensions

Chapter 20

Python When It Helps

This short chapter marks the start of the extension part of the textbook. Orange remains the main visual tool in the early chapters, but Python earns its place when the work needs reproducible code, automation, or libraries outside Orange's documented widget set.

Large language models also reshape this split: now that many data scientists prompt an LLM to draft code rather than writing every line by hand, the choice of programming language matters less than it once did, even when Python still dominates introductory examples.

20.1 When code earns its place

The tool split is practical rather than ideological.

- Use **Orange** when the goal is intuition, quick exploration, low-code workflows, and classroom discussion.
- Use **Python or other languages** when the work needs reproducible code, automation, version control, or methods that are outside Orange's documented widget set.

Neither tool is the "real" one and the other the "toy". They simply solve different teaching and engineering problems.

20.2 What notebooks, data files, models, and workflows save

The file distinction matters more once we start moving between GUI work and code.

- An Orange `.ows` file stores the **workflow**: which widgets you used, how they are connected, and the settings saved on the canvas.
- A dataset export stores the **data**: for example a cleaned CSV or tab-delimited table produced after filtering, selecting columns, or creating new features.

- A Python notebook or script stores the **code**: the steps needed to recreate the analysis in a programmable form.
- A saved model stores the **trained result**: the fitted object that can later score new data.

So when we say "save the workflow", we mean the Orange canvas. When we say "export the cleaned data", we mean a table another tool can read. When we say "save the model", we mean the fitted learner rather than the surrounding workflow. Those artefacts are related, but they are not interchangeable.

Chapter exercises

1. For five short scenarios, choose Orange, Tableau, Python, or a combination. Justify each choice using reproducibility, speed, stakeholder needs, and method availability.
2. Match notebooks, scripts, CSV files, saved model files, and Orange workflows to the part of the analysis they preserve.
3. Translate a simple Orange staging workflow into high-level Python steps. You do not need code yet; list the operations in the order a notebook would run them.

Chapter 21

Robust Regression in Python

Chapter overview

This short extension picks up the comparison deferred from chapter 8. The goal is not to crown a winner or to pretend that Python is automatically better. The goal is to keep the preprocessing fixed, swap only the estimator, and check whether robust methods actually earn their extra complexity.

21.1 Why Python earns its place here

Orange is excellent for teaching the baseline regression workflow, but scikit-learn makes it much easier to compare OLS, Theil–Sen, and RANSAC inside the same preprocessing pipeline. That lets us ask one fair question: if the cleaning and cross-validation stay fixed, does a robust estimator really improve the result?

21.2 A small, honest Python comparison

The comparison is easiest to read when we keep the preprocessing fixed and swap only the estimator. That makes the experiment fair: OLS, Theil–Sen, and RANSAC all see the same cleaned data and the same evaluation folds.

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import (
    LinearRegression,
    RANSACRegressor,
    TheilSenRegressor,
)
from sklearn.model_selection import cross_validate
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

```
numeric_features = ["land_size", "distance_cbd"]
categorical_features = ["suburb_ring"]

preprocess = ColumnTransformer(
    transformers=[
        ("num", Pipeline([
            ("impute", SimpleImputer(strategy="median")),
            ("scale", StandardScaler()),
        ]), numeric_features),
        ("cat", Pipeline([
            ("impute", SimpleImputer(strategy="most_frequent")),
            ("encode", OneHotEncoder(handle_unknown="ignore")),
        ]), categorical_features),
    ]
)

models = {
    "OLS": LinearRegression(),
    "Theil-Sen": TheilSenRegressor(random_state=42),
    "RANSAC": RANSACRegressor(random_state=42),
}

# X, y as prepared in the previous chapter
for name, estimator in models.items():
    model = Pipeline([
        ("preprocess", preprocess),
        ("regressor", estimator),
    ])
    scores = cross_validate(
        model,
        X,
        y,
        cv=5,
        scoring=[
            "neg_mean_absolute_error",
            "neg_root_mean_squared_error",
        ],
    )
    mae = -scores["test_neg_mean_absolute_error"].mean()
    rmse = -scores["test_neg_root_mean_squared_error"].mean()
    print(name, mae, rmse)
```

The important habit is comparison, not hero worship. If OLS and Theil–Sen are essentially tied, the simpler explanation may win. If RANSAC is better only because one obviously broken record is still in the data, the cleaner solution may be to fix the data source. Robust methods earn their place when they improve the analysis for the right reason.

21.3 Report the result plainly

A short report can be enough: “We began with ordinary linear regression. A small number of extreme redevelopment sales pulled the fitted line upward, so we compared OLS with Theil–Sen and RANSAC using the same cross-validation setup. The robust models reduced MAE on typical properties, so we kept the robust result for this valuation brief.”

This extension is the coded version of the earlier walkthrough. The hands-on activity in Part I stays deliberately simpler so readers can focus on diagnosis and model choice before they worry about library syntax.

Chapter exercises

1. Run the comparison for ordinary least squares, Theil–Sen, and RANSAC on the supplied small dataset. Record MAE and RMSE for each method.

2. Change the RANSAC residual threshold. Which observations move in or out of the inlier set, and how does the fitted line change?

3. Plot the fitted lines together. Explain which estimator resists outliers best and whether that resistance helps the actual question.

4. Write a short report that includes the chosen model, the evidence for that choice, and one data-quality warning.

Chapter 22

Probability Foundations for Data Science

This chapter sits in the extension part of the textbook because it adds formal probability language after the main Orange workflow story is already in place. We revisit the vocabulary of uncertainty, connect it to inference, and add the distributional ideas that later modelling chapters quietly rely on.

22.1 Interpretations of Probability

We compare frequentist and Bayesian viewpoints so our shared language for uncertainty remains consistent throughout this textbook.

Learning objectives

After reading this section, we will be able to:

- state the frequentist definition of probability as a long-run relative frequency;
- describe Bayesian probability as a quantified degree of belief that still obeys the same algebra;
- articulate when each interpretation provides intuition for real-world analytics problems.

Prerequisites

Before starting, make sure we can:

- compute simple relative frequencies from repeated experiments;
- read Venn diagrams or tree diagrams that represent joint and conditional events;

- explain in plain language what "uncertainty" means in the context of measurement or forecasting.

The frequentist perspective treats probability as the proportion of times an event occurs when we repeat an experiment indefinitely. When we say the probability of an online shopping cart being abandoned is 5%, we mean that over thousands of visits, about one in twenty carts never completes. This long-run framing is comfortable when we can imagine physically repeating the process—rolling dice, rerunning an A/B test, or simulating a customer journey under identical conditions. The vocabulary it gives us is grounded in counts, ratios, and the arithmetic of events, which is why it appears in introductory statistics courses and operational dashboards alike.

Bayesian probability broadens that language. Instead of demanding an infinite sequence of identical experiments, it lets us encode how strongly we believe an explanation fits the evidence. A product manager might judge there is a 70% chance that a new subscription feature will reduce churn even before a single user interacts with it, based on market research and previous launches. That belief is still disciplined by the same addition and multiplication rules we already rely on; the algebra of probability does not change. What changes is what the numbers represent: not just long-run frequencies, but coherent degrees of belief we intend to update when new data arrives.

Section summary

- Frequentist probability expresses uncertainty through long-run proportions observed over repeated experiments.
- Bayesian probability quantifies how strongly we believe an explanation, updating those beliefs as new evidence appears.

Self-check questions

1. When do repeated-trial arguments give us the clearest intuition for probability, and when do we prefer to reason in terms of belief?

Self-check reflections

1. Repeated-trial arguments shine when we can picture running the process many times, while belief-based language helps when we must make a call before those repetitions exist.

22.2 Connect Probability to Inference

Probabilistic reasoning lets us move from data to causes, preparing us for the modelling chapters that follow.

Learning objectives

After reading this section, we will be able to:

- distinguish between forward simulation (predicting outcomes) and inference (explaining observed data);
- use Bayes' rule conceptually to describe how new evidence revises our beliefs;
- outline how probabilistic thinking supports decision-making in analytic projects.

Prerequisites

Before starting, ensure we can:

- interpret conditional statements such as "the probability of A given B";
- summarise observational data sets with basic descriptive statistics;
- articulate a business or policy question that requires understanding why something happened, not just what will happen next.

Inference is about running probability in reverse. Instead of only forecasting what will happen if we know the data-generating process, we take the data we have and ask which underlying process is most plausible. Suppose a manufacturing line records a spike in defects. A purely forward model might simulate the number of defective units we expect tomorrow; an inferential analysis weighs competing explanations—material fatigue, operator error, or a calibration drift—and quantifies how likely each one is given the evidence collected. Probability becomes the currency of those arguments.

Bayes' rule formalises the reasoning. We start with prior beliefs about each explanation, perhaps informed by maintenance logs or earlier incidents. As new measurements arrive—sensor readings, visual inspections, or customer complaints—we assess how consistent those data are with each hypothesis. The rule tells us how to combine prior beliefs with the likelihood of the evidence to produce posterior beliefs (our after-evidence view of the world) that reflect both sources. Even when we do not plug numbers into the formula, the workflow guides our questions: What did we believe before? How surprising are the data under each candidate explanation? What should we believe now?

This inferential framing appears throughout the book. When we evaluate a clustering workflow in Orange, we are implicitly asking how credible each segmentation is given our observations. When we diagnose a regression model, we weigh whether residual patterns are more compatible with noise or with a missing variable. Working in a Bayesian frame keeps us honest about uncertainty, encourages us to seek additional evidence when the posteriors are diffuse, and equips us to explain to stakeholders why a recommendation is justified even when certainty is impossible.

Section summary

- Inference turns observed data into statements about underlying causes, rather than only predicting future outcomes.
- Bayes' rule provides the scaffold for updating beliefs as new evidence arrives.
- A probabilistic mindset supports transparent decision-making across analytics projects.

Self-check questions

1. How would we describe the difference between simulating tomorrow's demand and inferring why today's demand fell short?
2. In a recent project, what served as our prior belief and what evidence prompted us to revise it?

Self-check reflections

1. Simulation pushes probability forward to forecast outcomes, whereas inference runs it in reverse to explain the causes behind the data we already saw.
2. Name the baseline assumption we held, then the fresh evidence—such as new metrics, interviews, or experiments—that nudged us to update that belief.

22.3 Define Distributions, Random Variables, and Samples

We cement vocabulary around distributions, events, and random variables before heavier tools appear in later chapters.

Learning objectives

After reading this section, we will be able to:

- define a probability distribution and give examples from discrete and continuous settings;
- explain what makes a variable "random" in analytic practice;
- interpret histograms and sampled data as evidence of an underlying distribution.

Prerequisites

Before starting, make sure we can:

- read bar charts and histograms and describe what they show;
- distinguish categorical variables from numerical ones;
- recognise units of measurement relevant to our domain (such as dollars, minutes, or centimetres).

A **distribution** assigns probabilities to events. For a weekend football match we might describe the chances of a home win, a draw, or an away win; for customer support we might assign probabilities to response times falling within service-level agreements. Each distribution encodes the same idea: every mutually exclusive outcome gets a probability, and the list of probabilities sums to one. In discrete cases—dice, survey responses, or product categories—we can list those outcomes individually. In continuous cases—height, rainfall, or latency—we talk about ranges because the probability of any exact real number is effectively zero.

The next step is **random variables**. A random variable is any quantity whose value is governed by a distribution. We treat the daily number of checkouts, the temperature recorded at noon, or the proportion of positive reviews as random variables because, before observing them, each has a spread of plausible values.

Histograms make these definitions tangible. When we sample a random variable repeatedly—collecting customer satisfaction ratings for a month or measuring the fuel efficiency of vehicles in a fleet—we can plot the counts of observations falling into bins. Smooth histograms hint at the shape of the underlying distribution, revealing whether outcomes cluster, skew, or exhibit multiple modes.

Some named families are worth recognising early. Bernoulli and binomial distributions come from repeated yes/no counts. Poisson distributions describe rare-event counts. The normal distribution is the workhorse for additive noise. The log-normal family appears when positive quantities are driven by multiplicative effects. Cauchy-style heavy tails are the warning sign that not every distribution has a stable mean or variance, which is exactly why formal assumptions matter later.

Section summary

- Distributions map every possible event or range of values to a probability.
- Random variables are the measurable quantities we treat as governed by those distributions.
- Samples and histograms provide empirical glimpses of the underlying distributional shape.

Self-check questions

1. Which variables in our current project behave discretely, and which are better described with continuous distributions?
2. How does a histogram help us decide whether a modelling assumption, such as symmetry or unimodality, is reasonable?

Self-check reflections

1. Transaction counts or survey responses take discrete values, while timing, spend, or sensor readings usually fall on a continuous scale.
2. Histograms quickly show whether the data look balanced, skewed, or multi-peaked, helping us judge whether simple modelling assumptions fit or whether we need more flexible approaches.

22.4 Simulation Basics

We introduce sample data generators and histogramming so the law of large numbers becomes concrete.

Learning objectives

After reading this section, we will be able to:

- explain how sample data generators approximate random draws in software;
- describe how sample size affects the stability of estimated distributions;
- interpret simulations that illustrate the law of large numbers and small-sample volatility.

Prerequisites

Before starting, ensure we can read basic charts and follow what a simulation is trying to show.

For teaching, the visual part of the story can stay in Orange even when the draws themselves come from a prepared table or helper script. Once a column of sampled values is loaded, Orange's **Distributions** and plotting widgets let us compare what 100, 10,000, and 1,000,000 draws look like without burying the lesson inside code.

Simulations reveal how sample size shapes our understanding. Drawing one hundred values from a uniform distribution and plotting their histogram gives a noisy approximation of the underlying flat shape; repeating the exercise with ten thousand draws looks smoother,

and a million draws produces a histogram that visually hugs the theoretical line. This is the **law of large numbers** in action: as the number of independent samples increases, the sample average converges on the expected value, and sample histograms resemble the true distribution. At the same time, small-sample behaviour reminds us why caution is warranted. Tiny samples can produce streaks or gaps that disappear with more data, so we avoid over-interpreting early patterns.

A quick experiment cements the lesson. Generate or load uniform samples in three batches—10^2, 10^4, and 10^6 draws—then send them through the same Orange plotting workflow. Record how the variance of the sample mean shrinks and how the jagged edges smooth out. In each case we see the same trajectory: more data gives stability, and visualisations of the cumulative average settle around the true value.

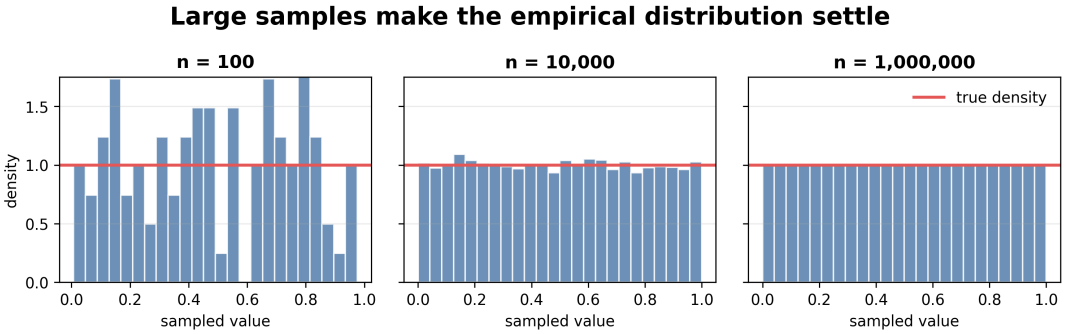


Figure 22.1: Uniform random draws look jagged in small samples and increasingly stable as the sample grows. The red line marks the true density the histograms are approaching.

Section summary

- Sample generators and Orange plots let us explore probabilistic ideas without hiding the picture inside code.
- Large samples make empirical summaries converge to their theoretical counterparts, demonstrating the law of large numbers.
- Small-sample volatility is expected, reminding us to temper conclusions when data are scarce.

Self-check questions

1. How many samples did we need in our own simulation before the histogram looked stable, and why?

Self-check reflections

1. Most simulations need thousands of draws before the shape settles; the volume smooths out random bumps that dominate tiny samples.

22.5 The Central Limit Theorem

We unpack why sums of draws trend normal, motivating the bell curve as a default model for noisy processes.

Learning objectives

After reading this section, we will be able to:

- state the central limit theorem (CLT) in practitioner-friendly language;
- describe the conditions under which sums or averages of random variables become approximately normal;
- connect the CLT to diagnostics and modelling choices used in later chapters.

Prerequisites

Before starting, make sure we can:

- compute means and standard deviations for sample data;
- explain the difference between summing and averaging observations;
- recognise normal distribution curves and their key features.

Adding independent random variables together smooths out their idiosyncrasies. Start with uniform draws between zero and one: the distribution of a single draw is flat, the sum of two looks triangular, and the sum of ten already resembles a bell curve. As we keep adding draws—or, equivalently, averaging them—the shape tends toward the normal distribution. The central limit theorem formalises that progression. It states that if we take independent draws from any distribution with a finite mean μ and variance σ^2, then the distribution of their average approaches a normal distribution with mean μ and variance σ^2/n as the sample size n grows.

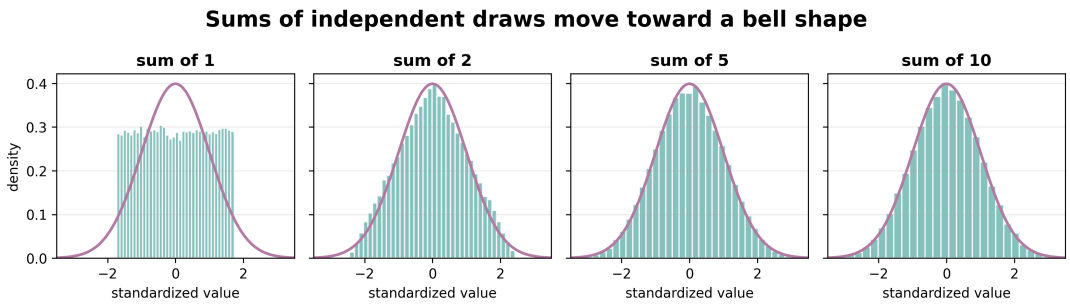


Figure 22.2: Sums of independent uniform draws move from a flat distribution toward an approximately normal shape. The purple curve shows the standard normal density for comparison after standardising each sum.

This result justifies the approximations we rely on in modelling and diagnostics. Confidence intervals, hypothesis tests, and control limits often assume normality not because the original data are bell-shaped, but because sums or averages of many observations inherit that shape through the CLT. When we evaluate model residuals or compute feature averages in later chapters, we implicitly lean on this behaviour to argue that our summary statistics have predictable variability.

Awareness of the theorem's boundaries keeps us cautious. Heavy-tailed distributions with infinite variance, serial dependence between observations, or extremely small sample sizes can all undermine the normal approximation. Store catalogue sizes, online follower counts, or other power-law-shaped business quantities are good reminders that enormous observations sometimes matter more than the nice bell-curve story suggests. In those cases we reach for alternative techniques—bootstrapping, transformation, or explicit time-series models—to respect the data's structure. Most of the time, though, the CLT gives us permission to proceed with normal-based reasoning.

One useful counterpoint is the **log-normal distribution**. Whenever a quantity is shaped by multiplying several positive factors together, the raw values often end up right-skewed while the logged values look much closer to normal. Product-related data often behave this way: basket values, item popularity, firm sizes, and some delivery-time measures can all be driven by compounding effects rather than neat additive noise. In practice that means a long right tail is not automatically a sign that the data are broken; it may be the natural footprint of a multiplicative process.

> **Computing extension — fitting an observed distribution.** In Python, a package such as `fitter` can compare candidate families against an observed sample and suggest whether the data look closer to a normal, log-normal, gamma, or some other distribution. Treat that output as a starting point rather than a verdict: the histogram, the domain story, and the sample size still matter. Orange does not foreground this style of distribution-fitting as directly, so Python is often the cleaner place to explore it.

Section summary

- The central limit theorem explains why sums and averages of many independent draws often look normal, regardless of the original distribution.
- Normal approximations underpin confidence intervals, residual diagnostics, and other analytical tools used in later chapters.
- Quantities driven by multiplicative effects often look log-normal, which is why a log transform can make business data easier to describe.
- Understanding the theorem's assumptions helps us recognise when to trust the approximation and when to reach for alternatives.

Self-check questions

1. Which workflow in our team averages many observations, and how does the CLT justify treating that average as approximately normal?
2. When might the CLT fail, and what alternative analysis strategies would we employ in those situations?

Self-check reflections

1. Averages built from many observations often settle into a bell-shaped pattern, so confidence intervals make sense.
2. The theorem struggles with heavy tails, dependence, or tiny sample sizes. In those cases, bootstrap resampling, robust summaries, or time-series models respect the structure better than a plain normal approximation.

Conclusion and next steps

Probability is not the first thing this textbook teaches, but later modelling chapters depend on it. Once the workflow story is familiar, these formal ideas are easier to place: they tell us when a summary is stable, when an assumption is fragile, and when a model's neat output deserves a second look.

Chapter exercises

1. Classify several uncertainty statements as frequentist, Bayesian, or informal probability language. Explain one borderline case.
2. For three data stories, identify the random variable, the observed sample, and the unknown quantity being estimated.
3. Simulate repeated draws from a simple distribution at three sample sizes. Describe how the histogram and sample mean change.
4. Compare sample means drawn from two different underlying distributions. What becomes more normal, and what does not?
5. Decide whether a long right tail in a business dataset suggests an error, an outlier, or a plausible multiplicative process.

Chapter 23

Explore NSW Road-Crash Data with pandas

Chapter overview

This chapter begins the optional computing-extension part of the textbook. If the early visual chapters have already given us enough, we can stop there and return later. If we want more code, we now turn the road-crash workbook into notebook-ready pandas work so our injury-risk modelling starts from a tidy, well-understood dataset. The chapter shows how to load the official New South Wales crash workbook, interrogate its schema, separate Series from DataFrames, and build engineered ratios that capture the difference between minor and life-threatening incidents. We finish by using Seaborn pair plots to surface correlated risk factors and point ahead to the later modelling chapters that reuse the same cleaned data.

23.1 Import and inspect the crash workbook

Use `pandas.read_excel` to load the crash records, establish a reliable index, and audit the incoming schema before modelling.

Learning objectives

After this section we will be able to:

- load large Excel workbooks directly into pandas using keyword arguments informed by the documentation;
- confirm basic structure with `.shape`, `.columns`, and `.head` so unexpected fields surface quickly;
- promote unique identifiers such as the crash number into an index for faster joins and clearer filtering.

Prerequisites

Before diving in, make sure we can:

- locate the crash-data workbook and upload files to Google Colab or a local Jupyter environment;
- interpret pandas `Series` and `DataFrame` outputs at a glance;
- recognise obvious missing-value sentinels (for example, impossible distances like 999,999 km).

Pandas fundamentals refresher

Before opening the workbook, pause to recall how pandas organises data. A `DataFrame` behaves like a spreadsheet table: rows represent records and columns hold variables, each with a shared index that labels the rows. A `Series` is the one-dimensional companion that stores a single column (or row) with the same index, making it ideal for targets or quick summaries. When we talk about the *index*, we mean that labelled axis—it can be the default integer range, a unique identifier such as the crash number, or a time stamp.

Pandas defaults sometimes hide this structure. Selecting a single column with square brackets drops the column dimension and returns a Series; wrapping the column name in a list preserves the DataFrame shape. Grouping, joins, and plotting all depend on keeping track of which form we are working with, so we will call it out each time we switch between them.

Because pandas leans on vectorised operations, we rarely write explicit loops. Instead we apply methods that operate on whole Series or DataFrames at once, which keeps analysis concise and performant. That design also means error messages often mention alignment, broadcasting, or data types; in the troubleshooting callout later in this chapter we decode the ones that show up most frequently in the crash workbook workflow.

Start by importing pandas alongside the plotting stack we will use later:

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

pd.options.display.max_columns = None
```

When we forget an argument, we can lean on the notebook help shorthand. Typing `pd.read_excel?` in Colab opens the docstring panel without executing the cell. With the path handy, load the crash sheet and promote the crash identifier to the index so each row mirrors a real-world incident:

```
df = (
    pd.read_excel(
```

```
        "nsw_road_crash_data_2019-2023_crash.xlsx"
    )
    .set_index("Crash ID")
)
```

Immediately check the shape and skim the first records. The crash workbook contains 93,491 rows and 50 columns, so a quick `df.shape` sanity-check reassures us the upload succeeded. Follow with `df.head()` and `df.columns` to confirm columns like `Degree of crash`, `Two-hour intervals`, `Speed limit`, `Road surface`, and `No. of traffic units involved` appear as expected. Using `df.describe()` on numeric fields highlights skew: some records have zero distances, while others use a high sentinel such as 999,999. Flag those anomalies in the notebook so the team agrees they represent missing or unusual reporting rather than ordinary distances.

Source note. The textbook example uses a teaching copy of NSW road crash records distributed through NSW government data channels. We use a reduced classroom workbook here, but the field definitions and crash categories follow the official NSW crash-data releases listed in the references chapter.

Pair `df.head()` with `df.info()` and `df.dtypes` to confirm data types before we engineer features; mismatched types often cause downstream joins to fail.

Accessibility spotlight. Share a textual summary alongside screenshots when briefing stakeholders: list the headline column counts, noteworthy null markers, and key categories so readers who rely on screen readers do not miss the context hidden in tables.

Section summary

- Use the inline help in notebooks to recall loader arguments on demand.
- Promote unique identifiers to the index immediately after loading; it simplifies joins later.
- Inspect shape, column names, and descriptive statistics before proceeding to ensure nothing silently dropped during import.

Self-check questions

1. Which pandas method reveals whether unexpected sheets exist in the workbook before we call `read_excel`?
   Hint: explore `pd.ExcelFile` to list sheet names.

2. How would we document the decision to treat 999,999 km as missing data so future analysts maintain the same cleaning rule?
   Hint: think about a project README or an inline comment near the imputation code.

Solutions and discussion

1. Use `pd.ExcelFile(...)` to open `nsw_road_crash_data_2019-2023_crash.xlsx`, then inspect `.sheet_names` to preview sheets. This avoids unexpected tabs derailing imports.

2. Record the assumption in two places: add a short note in the project README that explains the sentinel value and place an inline comment near the cleaning code (for example, where we replace 999,999 with `NaN`). Future collaborators can then confirm the rationale before they tweak the pipeline.

23.2 Separate Series from DataFrames when selecting features

Distinguish one-dimensional Series operations from multi-column DataFrame workflows so selections, summaries, and filters stay predictable.

Learning objectives

After this section we will be able to:

- split the crash dataset into feature and target frames using column lists or drop-based exclusions;
- call Series-specific helpers such as `value_counts()` (which summarises frequencies, optionally as proportions when `normalize=True` instructs pandas to show ratios instead of raw counts) to understand class balance;
- craft boolean masks (filter conditions that evaluate to `True` or `False` for each row) that filter DataFrames without losing the index alignment between inputs and targets.

Prerequisites

Ensure we can:

- interpret the difference between `df["column"]` (Series) and `df[["column"]]` (DataFrame);
- use Python lists to reference multiple column names reliably;
- explain why leaking outcome columns (for example, `No. killed`) into the feature set compromises evaluation.

Begin by isolating the target outcome. We are asking a binary question: did any injury occur? The workbook records separate counts for people killed, seriously injured, moderately injured, and minor-or-other injured. Selecting those columns with a list returns a DataFrame; summing across columns with `axis=1` gives one injury total per crash, and the comparison converts that total into a Series target:

```
injury_cols = [
    "No. killed",
    "No. seriously injured",
    "No. moderately injured",
    "No. minor-other injured",
]
injury_total = df[injury_cols].sum(axis=1)
y = (injury_total > 0).astype(int)
y.value_counts(normalize=True)
```

That quick tally captures the goal: before fitting anything clever, know how often the positive class appears. Logging the proportions early helps justify baseline models (for example, a "predict injury every time" dummy) before we celebrate more sophisticated scores.

For the feature matrix, pick the context columns that plausibly influence severity. One option is an explicit inclusion list:

```
feature_columns = [
    "Street type",
    "Speed limit",
    "Type of location",
    "Weather",
    "Key TU type",
    "First impact type",
]
X = df[feature_columns]
```

Alternatively, drop leakage columns by name to keep the rest of the schema. In this workflow we remove both the injury-count columns used to create `y` and the already-summarised severity fields:

```
X = df.drop(
    columns=injury_cols + [
        "Degree of crash",
        "Degree of crash - detailed",
    ]
)
```

Both strategies retain a DataFrame with the same index as `y`, preserving row alignment for modelling. Use boolean masks to focus on specific scenarios without copying data:

```
fatal = df[df["Degree of crash"] == "Fatal"]
night_rain = df[
    (df["Weather"] == "Rain") &
    (df["Two-hour intervals"].str.startswith(
        ("18", "20", "22", "00"), na=False
    ))
]
```

Because pandas maintains index alignment, we can join summaries (for example, average speed limit by region) back onto `X` without resorting to merges. Remember to use `.copy()` when creating derived slices we intend to mutate; this avoids the "SettingWithCopy" warnings that surface when chained indexing mutates a view.

Troubleshooting clinic. Keep these fixes nearby when selections misbehave:

- *Empty mask.* Print `df["Weather"].unique()` to check spacing or casing, then normalise with `.str.strip()` first and `.str.casefold()` second.
- *SettingWithCopy warnings.* Call `.copy()` on any slice we plan to mutate so pandas works on a fresh DataFrame instead of a view.
- *Unexpected type errors.* Inspect `df.dtypes` to confirm categorical columns have consistent types. Mixed integers and strings often appear when spreadsheets store placeholders such as "N/A".
- *Misaligned joins.* Reset or set the index explicitly on both frames before joining to ensure the alignment columns match; mismatched indexes introduce `NaN`s.

Section summary

- Treat targets as Series so we can leverage Series-specific utilities such as `value_counts()` and `map()`.
- Build feature DataFrames either by explicit inclusion lists or by dropping leakage columns; both patterns maintain index alignment.
- Compose boolean masks carefully and normalise categorical text before comparison to avoid silent mismatches.

Self-check questions

1. Why does `df[["Degree of crash"]]` return a DataFrame while `df["Degree of crash"]` returns a Series, and when would we prefer each form?
   Hint: think about integration with scikit-learn versus descriptive statistics.

2. How can we guard against accidental target leakage when we maintain a long drop-column list?
   Hint: consider centralising the list of forbidden columns in a shared constant or config file.

Solutions and discussion

1. Single brackets return a Series because pandas assumes we want the column without its table structure; double brackets keep the DataFrame shape. Reach for the Series when we need Series-only helpers such as `value_counts()` or `map()`, and keep the DataFrame when libraries like scikit-learn expect two-dimensional input.

2. Store the forbidden columns in a constant (for example, `PROTECTED_COLUMNS`) and reuse it wherever we build features. Centralising the list keeps the drop logic consistent and makes code review easier when new target-like fields appear.

23.3 Engineer ratios and visualise numeric relationships

Create interpretable features from the crash counts and use Seaborn pair plots to surface correlated risk factors.

Learning objectives

After this section we will be able to:

- derive ratio-style features (for example, people injured per vehicle) that distinguish high-severity crashes;
- handle division safely when counts include zeros or missing values;
- generate and interpret Seaborn pair plots on filtered subsets without overwhelming the notebook session.

Prerequisites

Confirm we can:

- use `.assign()` to add new columns to a DataFrame without mutating the original;
- recognise when to replace zeros with `NaN` before computing ratios;
- read scatterplot matrices and histograms to comment on monotonic or clustered patterns.

Crash reports bundle raw counts: number of traffic units involved, people killed, and counts of serious versus minor harm. Ratios help us reason about severity. For example, a two-unit incident with multiple serious injuries deserves prioritised investigation compared with a tow-away where no one was hurt. Build those derived signals defensibly:

```
# Clip traffic-unit counts to avoid div by zero
traffic_units = df["No. of traffic units involved"].clip(lower=1)

# Fill missing injury counts with zero
injured = df[injury_cols].sum(axis=1).fillna(0)
serious = df["No. seriously injured"].fillna(0)

# Compute injury ratios
inj_per_unit = injured / traffic_units
serious_rate = serious / traffic_units

# Add the new features to main dataframe
df["injuries_per_traffic_unit"] = inj_per_unit
df["serious_injury_rate"] = serious_rate
```

Clipping the denominator at one (`clip` limits values to the range we supply) avoids division-by-zero errors when records log zero traffic units, and filling missing counts with zero keeps the ratios grounded. Document the assumptions (for example, "missing injuries imply none reported") in the notebook so reviewers can challenge them if local reporting rules differ.

With engineered features in place, explore numeric relationships using Seaborn. Pair plots reveal which ratios cluster by severity, but rendering the full 93,491-row dataset is slow. Sample first to keep rendering responsive:

```
sample = df.sample(n=5_000, random_state=8)
numeric_view = sample[[
    "No. of traffic units involved",
    "No. killed",
    "No. seriously injured",
    "No. moderately injured",
    "No. minor-other injured",
    "injuries_per_traffic_unit",
    "serious_injury_rate",
]]

sns.pairplot(
    numeric_view, diag_kind="hist", corner=True)
plt.suptitle("Injury counts and ratios", y=1.02)
plt.show()
```

> **Computing extension.** Estimate the cost of these transformations. Counting operations like `value_counts()` run in roughly $O(n)$ time because they touch each row once, and a single pair-plot panel is also about $O(n)$ in the number of rows because each point is drawn once. The total work grows with the number of column pairs rendered, not with the square of the row count. Only certain operations—non-indexed joins or all-pairs computations—approach

$O(n^2)$. When datasets grow past a few hundred thousand rows, budget time for chunked loading or downsampling so notebooks stay responsive.

Expect to see tight clusters near zero (no injuries) and narrow bands where serious-injury rates climb with traffic-unit counts. Highlight those findings in margin notes so the later modelling work can pick up the story when fitting classifiers.

Try this variation. Run the same pair plot separately for major urban, minor urban, and non-urban records (use the `Urbanisation` column) to check whether the injury ratios diverge. A split like that can inform more targeted road-safety recommendations.

Section summary

- Ratios derived from raw counts often express severity better than absolute numbers; clip denominators and fill missing values deliberately.
- Sampling before plotting keeps Seaborn responsive while preserving the overall pattern of correlations.
- Use pair plots to validate intuitive relationships before investing time in more complex models.

Self-check questions

1. How would we adapt the ratio pipeline for crashes involving pedestrians, where the number of traffic units may legitimately be zero?
   Hint: consider a separate denominator that clips at one for traffic units but allows a non-zero count of people involved.

2. Which additional numeric columns (for example, `Speed limit`) would we add to the pair plot to test the "speed increases fatal risk" hypothesis?
   Hint: think about scaling or encoding categorical speed limits into integers first.

Solutions and discussion

1. Replace the denominator with a blended measure: keep `traffic_units_clipped` for vehicle-focused incidents but add a `people_involved` denominator that clips at one for pedestrian counts. Switch to that denominator when `No. of traffic units involved == 0` so we avoid inflating ratios while still capturing serious harm.
2. Enrich the pair plot with `Speed limit` converted to numeric form (for example, extract the number from labels such as `60 km/h`) and consider including an hour

derived from `Two-hour intervals`. These variables provide context for whether higher speeds or night-time conditions align with more severe outcomes.

23.4 Carry the cleaned data into modelling

Carry the cleaned, feature-rich DataFrame forward so later modelling can focus on evaluation rather than wrangling.

Spend a moment documenting the pipeline we assembled: list the import commands, the index promotion, the engineered ratios, and any filtering rules we applied (such as sampling or text normalisation). Save the notebook with those notes so we can reproduce the setup without replaying the entire walkthrough. Later notebook work will reuse `X`, `y`, and the engineered ratio columns as inputs to scikit-learn pipelines, closing the loop between exploratory pandas work and the deployment-focused pipelines in chapter 26.

Section summary

- Share reproducible notebooks that encapsulate both the code and the rationale for each cleaning step.
- Keep the engineered ratios alongside the raw counts; later work may compare model performance with and without them.
- Cross-reference the pipeline chapter so readers see how exploratory ratios feed into production-ready preprocessing.

Self-check prompt. Before moving on, rehearse explaining the wrangling steps aloud: how did we load the Excel file, which anomalies did we flag, how did we separate Series and DataFrames, and which ratios told the most compelling story about crash severity? Being able to narrate the workflow builds confidence when we return to the notebook later.

Chapter exercises

1. Load the crash workbook, inspect the sheet names and header rows, and produce a first clean DataFrame. Record the resulting row and column counts.
2. Predict whether several pandas selections return a Series or a DataFrame, then run them and correct our notes.
3. Engineer one severity ratio, filter to a defensible subset, and explain which rows were excluded.
4. Generate or inspect a pair plot. Write two cautious observations without claiming causation.
5. Save a cleaned CSV and record its row count, column count, and SHA-256 hash.

Chapter 24

Matplotlib Foundations for Exploratory Plots

Chapter overview

We turn an introductory plotting walkthrough into a reliable Matplotlib recipe that keeps our exploratory charts tidy and reproducible. The chapter covers the figure–axes mental model, scatter plot conventions, layout adjustments, and the shortcuts available through pandas. We finish with export patterns so every chart can move from the notebook into reports without pixelation.

24.1 Adopt the figure–axes workflow

Create the figure and axes explicitly so every plot call has a predictable canvas.

Learning objectives

After this section we will be able to:

- import `matplotlib.pyplot` using the common `plt` alias alongside pandas and NumPy;
- instantiate a figure and axes pair with `plt.subplots()` and reason about the tuple return signature;
- control the canvas size with the `figsize` argument and explain how on-screen scaling differs from print layouts.

Prerequisites

Make sure we can:

- read tabular data into a pandas `DataFrame`;

- run notebooks in Google Colab or a local Jupyter environment;
- interpret Python functions that return multiple values.

Matplotlib thinks in terms of a *figure*—the whole image—and one or more *axes* objects that host individual charts. Calling `fig, ax = plt.subplots(figsize=(6, 4))` returns both components so we can customise them directly. The tuple unpacking mirrors earlier tools such as `train_test_split`, grounding readers who have seen multi-value returns before.

The `figsize` argument uses inches; Matplotlib rescales the display to fit our notebook window but retains the specified proportions when we save to disk. On a laptop, a 20 inch width still shrinks to the available viewport, yet the exported PNG respects those physical dimensions. We can confirm the types with `type(fig)` and `type(ax)` if the tuple feels abstract at first; Matplotlib reports a `Figure` and an `Axes` (whose type prints as `matplotlib.axes._axes.Axes`) respectively, reassuring us that we have the expected handles for later methods.

To make the handles tangible, build two axes at once: one for an environmental snapshot and another for a workplace survey. The code below labels each axis according to the story it tells so we can reference them later when styling.

Example: paired scatter layout.

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

air_quality = pd.DataFrame({
    "pm25": [11.2, 13.4, 9.8, 15.1, 12.0],
    "tree_canopy": [28, 24, 35, 19, 31],
    "city": [
        "Harbour City", "River Junction",
        "Coastal Ridge", "Lakeside", "Sunset Hills"
    ],
})

commute = pd.DataFrame({
    "remote_days": [0, 1, 2, 3, 4],
    "median_commute": [62, 58, 54, 51, 49],
})

fig, (ax_air, ax_commute) = plt.subplots(
    1, 2, figsize=(10, 4)
)

ax_air.scatter(
    air_quality["pm25"],
    air_quality["tree_canopy"],
    s=air_quality["tree_canopy"] * 8,
```

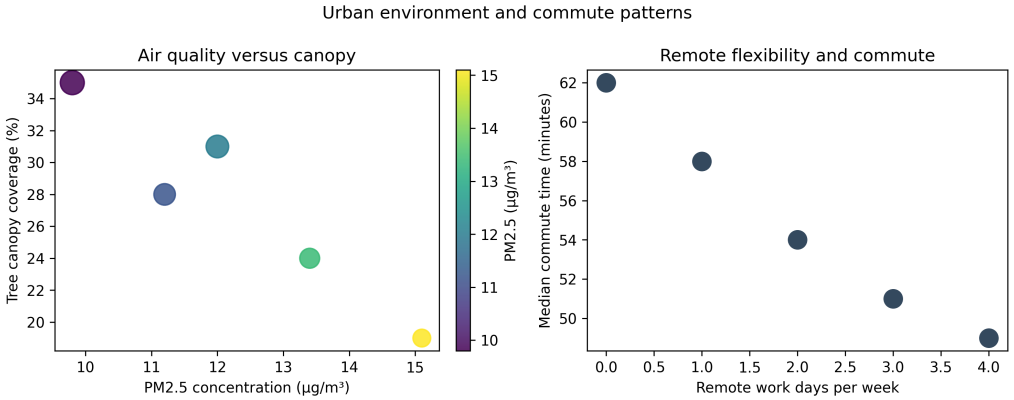


Figure 24.1: Two scatter plots generated from the same figure: the left axis compares particulate matter with tree canopy coverage across five cities, while the right axis charts remote-work days against median commute times. Marker size encodes canopy coverage, and all labels come from explicit axis methods.

```
    c=air_quality["pm25"],
    cmap="viridis",
    alpha=0.85,
)
ax_air.set(
    xlabel="PM2.5 concentration (µg/m³)",
    ylabel="Tree canopy coverage (%)",
    title="Air quality versus canopy",
)

ax_commute.scatter(
    commute["remote_days"],
    commute["median_commute"],
    color="#34495e",
    s=160,
)
ax_commute.set(
    xlabel="Remote work days per week",
    ylabel="Median commute time (minutes)",
    title="Remote flexibility and commute",
)

fig.suptitle("Environment and commute")
fig.tight_layout()
```

Running the snippet creates a figure with two clearly labelled axes, shown in Figure 24.1. Naming the axes variables `ax_air` and `ax_commute` makes subsequent styling read like prose, which is especially helpful when sharing notebooks with collaborators.

Communication tip. When explaining the figure–axes split verbally, summarise how each object contributes to the final chart. That narration helps readers picture the relationship before we dive into code.

Section summary

- Import pyplot as `plt` and unpack `plt.subplots()` into figure and axes variables every time.
- Treat `figsize` as a print-setting: it locks proportions for exports even if the notebook preview scales.
- Lean on Python's tuple unpacking intuition from earlier data-splitting utilities.

Self-check questions

1. Why does `plt.subplots()` return two objects, and how does that compare with `train_test_split`?
   Hint: think about keeping handles for later method calls.

2. How would you explain the difference between on-screen scaling and the saved figure size to a teammate preparing print assets?
   Hint: relate pixels in the browser to the inch-based `figsize` setting.

24.2 Build scatter plots with axis methods

Use the axes handle to plot, label, and adjust markers with confidence.

Learning objectives

After this section we will be able to:

- call `ax.scatter(x, y, s=...)` to create dot plots anchored to our axes;
- control marker sizing with the `s` argument and reason about how extreme values affect readability;
- label axes and titles with `set_xlabel`, `set_ylabel`, and `set_title`;
- tighten layouts with `fig.tight_layout()` to avoid clipped text when multiple plots share a figure.

Prerequisites

Before proceeding we should:

- assemble a tidy DataFrame containing the numeric columns we intend to plot;
- recall Python dictionary-to-DataFrame construction for quick examples;
- understand basic scatter-plot interpretation (cluster spread, trends).

Start with the air-quality axis from Figure 24.1. Because the DataFrame tracks particulate matter, tree canopy, and city names, we can encode canopy coverage as marker size and colour each dot by its PM2.5 reading. That single call to `ax_air.scatter` communicates three variables at once without overwhelming readers. Rotate to the commute dataset on the right-hand axis to illustrate a completely different use case: smaller, uniformly coloured markers that highlight the downward trend between remote-work flexibility and travel time. Having two contexts in the same figure reinforces that the axes workflow adapts to environmental monitoring, workforce analytics, or any other domain we work with.

Once `ax.scatter` draws the points, experiment with `s`: small values such as 1 produce hairline dots suited to dense datasets, while large values like 10 000 swallow nearby observations. Label axes immediately after plotting; otherwise presentations inherit default column names that may confuse stakeholders. Call `ax.legend()` whenever multiple series share an axis so readers can distinguish the markers. The same pattern applies to `ax.plot` for line charts: the axes handle keeps styling consistent whether we draw dots, connected trends, or overlays. A concise title reinforces the narrative focus (for example, *Remote flexibility and commute*). Conclude with `fig.tight_layout()` to ask Matplotlib to resize subplots and padding automatically—a guard rail when we add more axes later.

Troubleshooting clinic. If column names contain spaces or apostrophes, copy them carefully or reference them via `df["Tree canopy"]` to avoid syntax errors. A `KeyError` often signals a stray capital letter, so compare `df.columns` against the strings passed into `x` and `y`. When labels overlap despite `tight_layout`, rotate tick labels with `ax.tick_params(axis="x", labelrotation=45)` so values remain legible. If a `TypeError` complains about non-numeric data, convert the columns with `pd.to_numeric(errors="raise")` so unexpected strings surface early.

Performance note. Large geospatial or telemetry datasets can overwhelm scatter plots. Consider plotting a stratified sample, aggregating to hourly or daily summaries, or enabling rasterisation with `ax.scatter(..., rasterized=True)` so exports stay responsive. Combining `alpha=0.5` with smaller marker sizes reveals dense clusters without freezing the notebook.

Presentation tip. Summarise each scatter in plain language (for example, “Dark-blue circles trend upward as ages increase, showing taller individuals in older groups”) so the main trend is clear even before anyone inspects the code.

Section summary

- Plot through the axes handle to keep all styling in one place.
- Moderate `s` values so dots stay visible without smothering neighbours.
- Always apply layout tightening before exporting or sharing the notebook.

Self-check questions

1. How would you justify a chosen `s` value when communicating with a non-technical audience?
   Hint: tie marker size to the story you want the trend to tell.

2. When should you choose axis method calls over letting pandas infer labels for you?
   Hint: consider how much control you need over titles, colours, and later annotations.

24.3 Resize figures and export publication-ready images

Treat figure size, DPI, and file formats as part of the modelling workflow.

Learning objectives

After this section we will be able to:

- resize figures deliberately for notebooks versus print media;
- save figures via `fig.savefig()` with `dpi` and `bbox_inches` to preserve layout;
- choose between PNG and PDF outputs based on downstream usage;
- use `pathlib.Path` to build cross-platform export paths.

Prerequisites

We should already know how to:

- navigate the Colab or local file browser;
- recognise common raster versus vector file formats;
- manage Python imports from the standard library.

Doubling `figsize` values (for example, `figsize=(12, 8)`) creates more breathing room for annotations, yet the notebook preview may still shrink the chart to fit the browser pane. Shrinking `figsize` has the opposite effect: fonts appear enormous because the same pixel budget represents a physically smaller canvas. Discuss these trade-offs with collaborators before finalising dimensions for slides or print.

When the layout looks right, export through `fig.savefig` instead of right-clicking the inline image. Use `pathlib.Path` to build the output path, set `dpi=300` for sharp printouts, and use `bbox_inches="tight"` to trim excess whitespace. Swap the extension for `.pdf` when vector artwork suits the destination (for example, design software or LaTeX). In hosted environments such as Colab, remind teams that files land in the session storage; download them before the runtime resets.

Section summary

- Match `figsize` and `dpi` to the medium where the figure will live.
- Prefer `fig.savefig` over manual downloads to guarantee repeatable exports.
- Build file paths with `pathlib` so the script runs cleanly on any operating system.

Self-check questions

1. Which parameters would you tweak when preparing a figure for print in a journal versus a slide deck?
   Hint: focus on `figsize`, `dpi`, and `bbox_inches`.

2. How can `pathlib` simplify collaboration across Windows, macOS, and Linux machines?
   Hint: consider how it abstracts away backslash versus forward-slash differences.

24.4 Leverage pandas plotting shortcuts

Choose between axes methods and DataFrame helpers depending on how much automation you need.

Learning objectives

After this section we will be able to:

- call `df.plot.scatter(x=..., y=..., ax=ax)` as a concise alternative;
- explain how pandas infers axis labels from column names and when to override them;
- weigh the trade-offs between brevity and explicit control when choosing a plotting API.

Prerequisites

Before adopting the shortcut we should:

- understand keyword arguments in pandas plotting methods;
- be comfortable passing Matplotlib axes into helper functions;
- recognise when automatic labelling may expose unfriendly column names.

Example: power-grid planning with pandas.

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

energy = pd.DataFrame({
    "solar_share": [0.18, 0.25, 0.32, 0.41, 0.29],
    "evening_demand": [420, 390, 370, 355, 400],
    "region": [
        "Coastal Belt", "Highland", "River Plains",
        "Sun Corridor", "Northern Range"
    ],
})

fig, ax = plt.subplots(figsize=(6, 4))
energy.plot.scatter(
    x="solar_share",
    y="evening_demand",
    s=energy["evening_demand"],
    color="#d35400",
    ax=ax,
)

for _, row in energy.iterrows():
    ax.annotate(
        row["region"],
        (row["solar_share"],
         row["evening_demand"]),
        textcoords="offset points",
        xytext=(6, -10),
    )

ax.set_title("Solar vs evening peak demand")
ax.set_xlabel("Share from solar")
ax.set_ylabel("Evening demand (MW)")
fig.tight_layout()
```

Pandas delegates to Matplotlib under the hood, so `df.plot.scatter` reaches the same visual outcome with fewer lines of code. Because the DataFrame already knows about its

columns, we only supply the `x` and `y` column names plus the target axes. Labels default to the column headers, saving time during rapid exploration. When presenting findings, we still favour explicit `set_xlabel` calls to translate shorthand like `solar_share` into audience-ready phrasing.

Choosing between approaches hinges on intent. The axes-first pattern centralises styling decisions and keeps the Matplotlib API visible to newcomers. The pandas shortcut prioritises velocity when exploring dozens of combinations in a notebook. Start with the explicit approach, then switch to the concise form once the keyword arguments feel familiar.

Collaboration tip. Commit both versions of the plotting helper to version control and note the expected arguments in a docstring. That way, teammates reviewing a pull request can swap between the explicit axes workflow and the pandas shortcut without rewriting surrounding analysis cells.

Section summary

- Use pandas plotting helpers for quick experimentation, but keep axes handles for consistent styling.
- Override inferred labels before sharing outputs beyond the notebook.
- Remember that both APIs rely on Matplotlib, so knowledge transfers between them.

Self-check questions

1. When does the pandas shortcut save you time, and when might it hide important styling options?
   Hint: balance exploratory speed against presentation polish.

2. How could you refactor a notebook so teammates can choose either plotting style without rewriting code?
   Hint: wrap plotting logic in helper functions that accept an `ax` argument.

24.5 Connect Matplotlib workflows to Tableau dashboards

Spot the similarities so moving to Tableau feels familiar rather than foreign.

Learning objectives

After this section we will be able to:

- recognise how Matplotlib's figure–axes model maps to Tableau worksheets and dashboards;
- describe when to start in Matplotlib for programmable control versus Tableau for drag-and-drop exploration;
- prepare exports with Tableau in mind so colleagues can blend notebook and BI assets.

Prerequisites

Before making the jump we should:

- feel comfortable exporting figures from Matplotlib;
- know the basics of launching Tableau and connecting to a CSV file;
- understand the audience expectations for an interactive dashboard versus a static report.

Tableau's worksheet is conceptually close to a single Matplotlib axes: each hosts one visual that we configure through marks, encodings, and annotations. A Tableau dashboard mirrors a multi-axes Matplotlib figure where we compose several plots into a single view. Translating a scatter from Matplotlib to Tableau becomes easier when we articulate the encoding choices explicitly—what data fields drive the x- and y-positions, how colour groups categories, and which tooltips we rely on for context. Explaining those ingredients up front lets a teammate rebuild the chart in Tableau.

Deciding where to begin depends on our constraints. Matplotlib shines when we need version-controlled scripts, automated report generation, or statistical overlays that go beyond Tableau's default calculations. Tableau, on the other hand, accelerates collaborative exploration and interactive filtering during stakeholder workshops. Many teams prototype in Tableau to surface interesting slices, then translate the final story into Matplotlib for publication alongside the code that sourced the data.

When handing work between tools, plan the exports deliberately. Saving a high-DPI PNG or vector PDF from Matplotlib provides assets we can drop into Tableau dashboards as static reference panels. Likewise, exporting Tableau visuals as images or embedding them via Tableau Public can accompany Matplotlib plots in a shared report. Keeping filenames and captions consistent across both tools helps readers follow the same chart from one platform to the other.

Section summary

- Think of Tableau worksheets as cousins of Matplotlib axes, and dashboards as composed figures.
- Choose Matplotlib for scripted control and Tableau for interactive exploration depending on stakeholder needs.

- Coordinate exports so figures remain consistent when they move between notebooks and dashboards.

Self-check questions

1. How would you brief a teammate to rebuild your Matplotlib scatter in Tableau without losing key encodings?
   Hint: spell out the roles of position, colour, and annotations.

2. Which scenarios demand Tableau's interactivity, and when does Matplotlib's scripting pay dividends instead?
   Hint: contrast collaborative workshops against automated reporting pipelines.

Chapter wrap-up

We now have a dependable template for Matplotlib work: import pyplot with confidence, create figures and axes explicitly, build scatter plots with thoughtful sizing, and export the results through code. With the pandas helper in our back pocket, we can alternate between deliberate chart crafting and quick exploratory passes. The Tableau bridge keeps interactive BI within reach, positioning us to combine scripted visuals with interactive dashboards.

Chapter exercises

1. Create a labelled line plot using explicit `fig, ax` calls. Include axis labels, a title, and a legend where appropriate.

2. Build a scatter plot with labelled axes, sensible marker size, and a caption explaining what each visual encoding means.

3. Export the same figure at two sizes or resolutions. Explain which output is more appropriate for print and why.

4. Recreate one plot with a pandas plotting shortcut. What control did you gain or lose compared with direct axis methods?

5. Decide whether a stakeholder version of the same visual should be made in Tableau or Matplotlib. Explain the constraint driving the choice.

Chapter 25

Matplotlib Styling and Comparisons

Chapter overview

Building on chapter 24, we refine our visualisations with intentional colour palettes, marker shapes, bar charts for categorical comparisons, and trend-line overlays for regression problems. The chapter emphasises accessibility considerations and the interpretive cautions that prevent misleading stakeholders.

25.1 Select colours and markers for clarity

Choose palettes that honour colour-vision diversity and print constraints, and use markers to distinguish categories when hue alone is insufficient.

Learning objectives

After this section we will be able to:

- define custom colour palettes with hexadecimal codes and reason about RGB composition;
- map categorical values to colours using pandas `.map()` with a dictionary;
- apply matplotlib `cmap` objects (`plt.cm.Set2`, `viridis`, etc.) to retrieve perceptually uniform colours;
- distinguish when to use perceptual/sequential colour maps for continuous data versus qualitative colour maps for distinct categories;
- use marker codes (`o`, `x`, `^`, `s`) to layer a second categorical dimension on top of colour;
- explain colour-vision deficiency considerations and monochrome print fallbacks.

Prerequisites

Make sure we can:

- filter a DataFrame with boolean indexing (`df[df["species"] == "setosa"]`);
- apply the `.map()` method to transform a pandas Series via a lookup dictionary;
- recall the figure–axes workflow from chapter 24.

25.1.1 Build a colour palette dictionary

Colour often communicates category membership more directly than position or shape. A hexadecimal string like `"#d9534f"` tells Matplotlib how much red, green, and blue power to mix: `0xd9` (approximately 217) out of 255 for red, `0x53` (approximately 83) for green, and `0x4f` (approximately 79) for blue, yielding a warm red-orange. Gathering three or four such strings into a dictionary keyed by species, product tier, or risk level creates a consistent palette we can reuse across subplots.

Example: custom iris palette.

```
from sklearn.datasets import load_iris
import pandas as pd
import matplotlib.pyplot as plt

raw = load_iris(as_frame=True)
iris = raw.frame.rename(columns={"target": "species"})

name_dict = dict(enumerate(raw.target_names))
iris["species"] = iris["species"].map(name_dict)

palette = {
    "setosa":     "#1b9e77",
    "versicolor": "#d95f02",
    "virginica":  "#7570b3",
}

colors = iris["species"].map(palette)
```

Running `colors.head()` yields a pandas Series filled with hexadecimal strings aligned to each row's species label. That series then feeds into `ax.scatter(c=colors)` so every setosa dot renders in teal, every versicolor dot in orange, every virginica dot in purple.

25.1.2 Leverage matplotlib colour maps

Hard-coding colours becomes tedious when we have more than a handful of categories or when we want to respect accessibility standards. Matplotlib bundles dozens of colour maps organised by data type:

Perceptual / sequential maps such as `viridis`, `plasma`, or `cividis` transition smoothly from dark to bright, ideal for encoding continuous quantities (temperature, count, probability). They remain distinguishable when printed in greyscale and support many forms of colour-vision deficiency.

Qualitative maps such as `Set2`, `Paired`, or `tab10` provide distinct colours for unordered categories—iris species, product lines, experimental conditions. Matplotlib designers ensure adjacent colours differ in both hue and brightness, mitigating red–green confusion.

Example: using a qualitative colour map.

```
cmap = plt.cm.Set2

palette = {
    "setosa":     cmap(0),
    "versicolor": cmap(1),
    "virginica":  cmap(2),
}
```

Each `cmap(i)` call returns a four-tuple: red, green, blue, and alpha in $[0, 1]$. If we print `palette["setosa"]`, we see something like `(0.40, 0.76, 0.64, 1.0)`, representing 40% red, 76% green, 64% blue, and full opacity. Matplotlib accepts both tuple and hexadecimal formats interchangeably, so the scatter method interprets either correctly.

25.1.3 Combine colour with marker shapes

When readers view plots on a black-and-white printer or live with colour-vision deficiency, relying solely on colour can obscure category boundaries. Adding distinct markers (filled circles `o`, crosses `x`, triangles `^`, squares `s`) provides a redundant cue.

Example: species overlay with markers.

```
fig, ax = plt.subplots(figsize=(8, 6))

iris[iris["species"] == "setosa"].plot.scatter(
    x="sepal length (cm)", y="petal length (cm)",
    ax=ax, label="Setosa", color="red", marker="o"
)
iris[iris["species"] == "versicolor"].plot.scatter(
    x="sepal length (cm)", y="petal length (cm)",
    ax=ax, label="Versicolor", color="green", marker="x"
)
iris[iris["species"] == "virginica"].plot.scatter(
    x="sepal length (cm)", y="petal length (cm)",
    ax=ax, label="Virginica", color="blue", marker="^"
)
```

```
ax.legend()
fig.tight_layout()
```

Readers who cannot distinguish red from green still see the difference between a filled circle and a cross. That layered encoding respects the principle of perceptual redundancy: use shape *and* colour to communicate the same category membership.

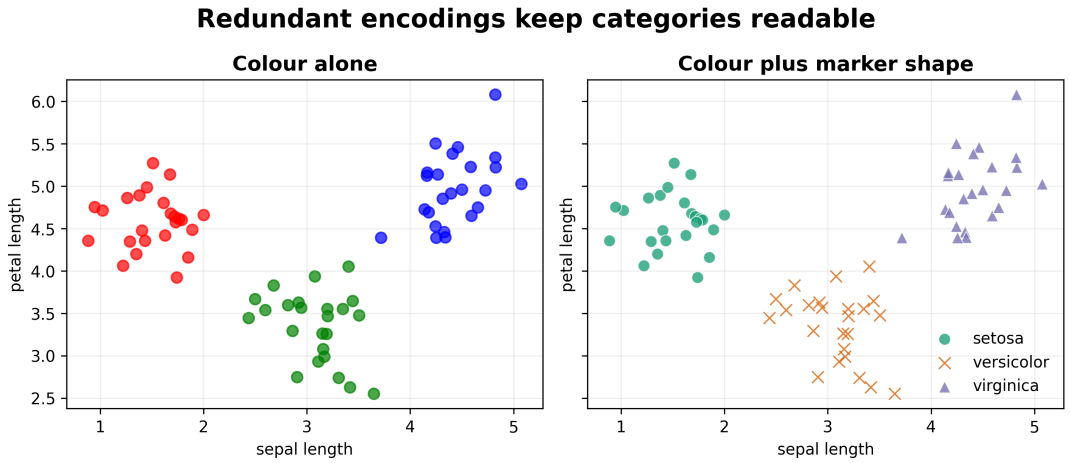


Figure 25.1: Colour alone can be fragile. Combining colour with marker shape gives readers a second cue when printing, projection, or colour-vision differences make hue harder to distinguish.

25.1.4 Adjust transparency and edge colour to prevent overlap

Dense scatter plots benefit from partial transparency (`alpha` < 1.0) so overlapping markers reveal density hotspots. Adding a white edge colour (`edgecolors="white"`) with a small line width (`linewidths=0.5`) carves a subtle boundary around each marker, making stacked dots easier to parse.

Example: transparency and edges.

```
ax.scatter(
    iris["sepal length (cm)"], iris["petal length (cm)"],
    c=colors, s=70, alpha=0.6,
    edgecolors="white", linewidths=0.5
)
```

Setting `alpha=0.1` yields near-invisible markers; `alpha=1.0` gives fully opaque dots. Most exploratory plots land between 0.5 and 0.8 to balance visual pop with overlap disclosure. Too-large marker sizes (`s=200` or more) often obscure the data plane itself, so iterate until the chart remains legible when several dots coincide.

25.1.5 Accessibility and print considerations

Around 8% of men and 0.5% of women experience red–green colour-vision deficiency. That prevalence means plots relying on pure red versus pure green risk excluding a significant

fraction of the audience. The following strategies mitigate such issues:

- **Brightness variation:** choose colours that differ in lightness (one pale, one dark) so greyscale conversion still separates them.
- **Perceptual colour maps:** `viridis`, `cividis`, and similar maps incorporate brightness ramps that preserve contrast when printed in monochrome.
- **Shape redundancy:** as demonstrated above, markers encode the same category that colour expresses.
- **Validation tools:** online simulators such as *Coblis* or browser plug-ins allow us to preview how a plot appears under deuteranopia (green-weak), protanopia (red-weak), and tritanopia (blue-weak) conditions. Use these before sharing with broader audiences.

When preparing figures for journal submission or conference proceedings, confirm that the publisher's monochrome reproduction retains clarity. If the venue only permits black-and-white printing, lean on line styles, marker shapes, and annotations rather than colour alone.

Section summary

- Define custom palettes with hexadecimal RGB codes or extract colours from matplotlib colour maps.
- Map categorical values to colours using `.map()` on the relevant pandas Series.
- Layer marker shapes on top of colour to support colour-vision diversity and monochrome printing.
- Moderate `alpha` and add edge colours to clarify overlapping markers.
- Test plots under colour-vision deficiency simulators and greyscale previews before publication.

Self-check questions

1. Why does combining both colour and marker shape improve accessibility, and what fraction of the population might benefit from that redundancy?
   Hint: recall the red–green colour-vision deficiency statistics.

2. How would you decide whether to use a perceptual colour map (like `viridis`) or a qualitative colour map (like `Set2`) for a given dataset?
   Hint: distinguish continuous numeric variables from discrete categories.

3. What `alpha` value would you choose for a scatter plot with 10 000 points concentrated in one region, and why?
   Hint: consider how overlapping opaque markers obscure density patterns.

25.2 Build bar charts for categorical comparisons

Switch from scatter to bar when categories lack natural ordering and we want to emphasise magnitude differences.

Learning objectives

After this section we will be able to:

- call `df.plot.bar(x="category", y="value")` to produce vertical bar charts;
- rotate x-axis tick labels to prevent label collisions when category names are lengthy;
- interpret bar heights as direct magnitude readings rather than position along a regression curve;
- distinguish when a scatter plot (regression context) versus a bar chart (classification or categorical comparison) best serves the data story.

Prerequisites

Before proceeding we should:

- assemble a DataFrame with one column holding category labels and another holding numeric magnitudes;
- recall the figure–axes workflow to control bar-chart styling;
- recognise classification problems (predict yes/no, species, tier) as distinct from regression problems.

25.2.1 Plot categorical data with `.plot.bar()`

Scatter plots excel when both axes represent continuous quantities and we expect a trend. Bar charts shine when the x-axis lists unordered categories (project phases, product brands, risk levels) and the y-axis measures a quantity (hours spent, revenue, incident count). Because category order is often arbitrary (we can list setosa before versicolor or vice versa), bars communicate membership and magnitude without implying a linear relationship.
Example: project-phase hours.

```
categories = pd.DataFrame({
    "category": ["Data prep", "Analysis", "Reporting", "Review"],
    "hours":    [12, 18, 9, 6],
})

fig, ax = plt.subplots(figsize=(8, 6))
categories.plot.bar(x="category", y="hours", ax=ax)

ax.set_ylabel("Hours spent")
ax.set_xlabel("Project phase")
ax.set_title("Time allocation by phase")
fig.tight_layout()
```

Each bar's height encodes effort. Because "Data prep" and "Analysis" sit side by side, we can instantly compare the two without mentally interpolating a trend line. If we had used a scatter plot, readers might search for a slope where none exists. The bar chart prevents that confusion by omitting the illusion of continuity.

25.2.2 Rotate tick labels to prevent overlap

Project-phase names such as "Requirements gathering" or "Stakeholder consultation" can overflow the x-axis if rendered horizontally. Rotating labels at a 45-degree angle reclaims that space and keeps text legible.

Example: label rotation.

```
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=45,
    ha="right"
)
```

`ha="right"` aligns the right end of each text label with the tick mark, improving visual consistency. If the rotation feels too steep, try `rotation=30` or adjust `figsize` to allocate more horizontal room. Some plots benefit from vertical labels (`rotation=90`), though that can reduce readability if category names are short.

25.2.3 Interpretive caution: bar charts and zero baselines

Because bar length communicates magnitude directly, omitting the zero baseline or truncating the y-axis distorts comparisons. A bar that appears twice as tall as another should represent twice the quantity, not simply a two-unit difference starting at an elevated baseline. Always confirm that the y-axis begins at zero unless the domain context explicitly justifies a non-zero start (for example, temperature anomalies anchored at a reference level).

Similarly, resist the temptation to order bars alphabetically if a more meaningful sequence exists. Sorting by magnitude (largest to smallest) or by a natural progression (project phases in chronological order) guides readers' eyes through the narrative more coherently than arbitrary alphabetical arrangement.

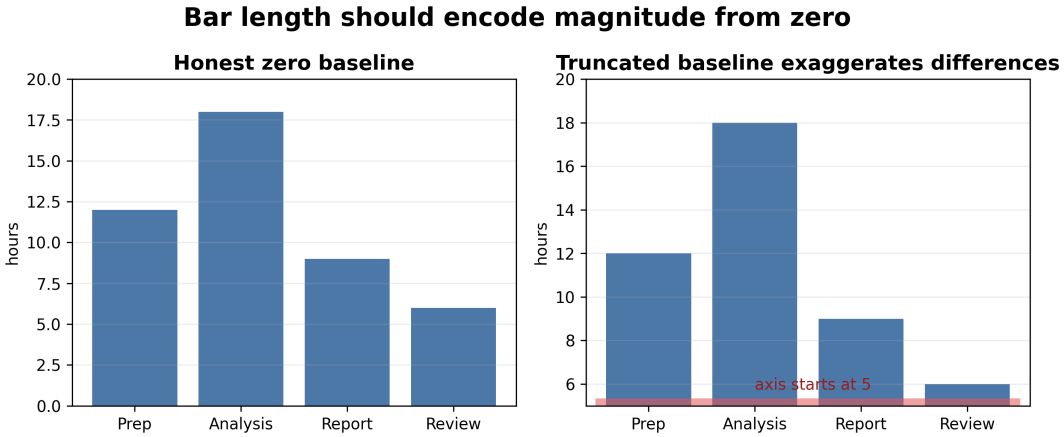


Figure 25.2: For bar charts, truncating the y-axis makes ordinary differences look much larger than they are. A zero baseline keeps bar length tied to magnitude.

Section summary

- Use bar charts when x-values are unordered categories and the y-axis measures magnitude.
- Rotate tick labels to prevent long category names from colliding on the x-axis.
- Anchor bar charts at zero to preserve honest magnitude comparisons.
- Sort bars by a meaningful criterion (size, time, domain logic) rather than alphabetically.

Self-check questions

1. Why is a scatter plot inappropriate for comparing hours spent across unordered project phases, and what misleading impression might it create?
   Hint: consider whether readers expect a trend line between adjacent categories.

2. Explain why starting a bar chart's y-axis at 10 instead of zero could mislead stakeholders about relative differences.
   Hint: think about how bar length encodes magnitude.

25.3 Overlay trend lines on scatter plots

Fit a linear-regression model and plot the predicted line atop the observed data to reveal relationships and residual patterns.

Learning objectives

After this section we will be able to:

- fit a `sklearn.linear_model.LinearRegression` model to a DataFrame column;
- construct a prediction DataFrame holding the minimum and maximum x-values to define the trend-line endpoints;
- call `ax.plot()` with the predicted values to draw the regression line over the scatter;
- style the trend line with dashed or coloured markers to distinguish it from raw data points;
- interpret the visual fit and residual spread when assessing model adequacy.

Prerequisites

Before proceeding we should:

- understand simple linear regression (slope, intercept) from earlier modelling chapters;
- filter or aggregate data into a DataFrame with numeric predictor and target columns;
- recall the figure–axes workflow to layer multiple plot calls on the same axis.

25.3.1 Fit a linear-regression model

When temperature and attendance exhibit a roughly linear relationship, fitting a regression line quantifies that trend and provides a visual reference for how individual observations deviate. Start by importing `LinearRegression` from `sklearn.linear_model`, fitting the model on the `temp` column to predict `attendance`, and extracting the slope coefficient and intercept.

Example: temperature versus attendance.

```
from sklearn.linear_model import LinearRegression

temps = pd.DataFrame({
    "temp":       [15, 17, 19, 20, 21, 23, 24],
    "attendance": [38, 42, 45, 50, 53, 58, 63],
})

lr = LinearRegression()
lr.fit(temps[["temp"]], temps["attendance"])

print(lr.coef_)       # approximately [2.76]
print(lr.intercept_)  # approximately -4.91
```

The coefficient ≈ 2.76 tells us that each one-degree increase in temperature predicts roughly three more attendees. The intercept ≈ -4.91 is only a mathematical anchor: taken literally it predicts negative attendance at zero degrees, a temperature far outside the observed data, so we should not read any real-world meaning into it. We can inspect `lr.coef_` and `lr.intercept_` to confirm the model trained correctly before plotting.

25.3.2 Construct a prediction DataFrame

To draw the regression line, we need two points: the predicted attendance at the minimum temperature and at the maximum temperature. Create a small DataFrame holding those extremes, pass it to `lr.predict()`, and capture the resulting array.

Example: prediction endpoints.

```
temp1 = temps["temp"].min()
temp2 = temps["temp"].max()

pred_frame = pd.DataFrame({"temp": [temp1, temp2]})
predictions = lr.predict(pred_frame)

print(predictions)  # [36.5, 61.3] approximately
```

`predictions` now holds the y-coordinates at `temp1` and `temp2`. Those two points anchor the line we will draw.

25.3.3 Plot the trend line over the scatter

With prediction endpoints in hand, call `ax.plot()` using the prediction-frame temperatures as x-values and the predicted attendances as y-values. Matplotlib connects those two points with a straight line, overlaying it atop the earlier scatter. Choose a dashed line style and a distinct colour to avoid confusing the fitted line with raw observations.

Example: overlaying the regression line.

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(temps["temp"], temps["attendance"], label="Observed")
ax.plot(pred_frame["temp"], predictions, color="red", linestyle="--",
↪  label="Fit")

ax.set_xlabel("Temperature (°C)")
ax.set_ylabel("Attendance")
ax.set_title("Temperature versus attendance with linear fit")
ax.legend()
fig.tight_layout()
```

Readers can now compare the fitted slope against individual points, spotting outliers or non-linear curvature that a simple line might miss. If observations cluster tightly around the dashed line, the linear model provides a reasonable summary; if they diverge widely, consider polynomial features, transformations, or robust regression.

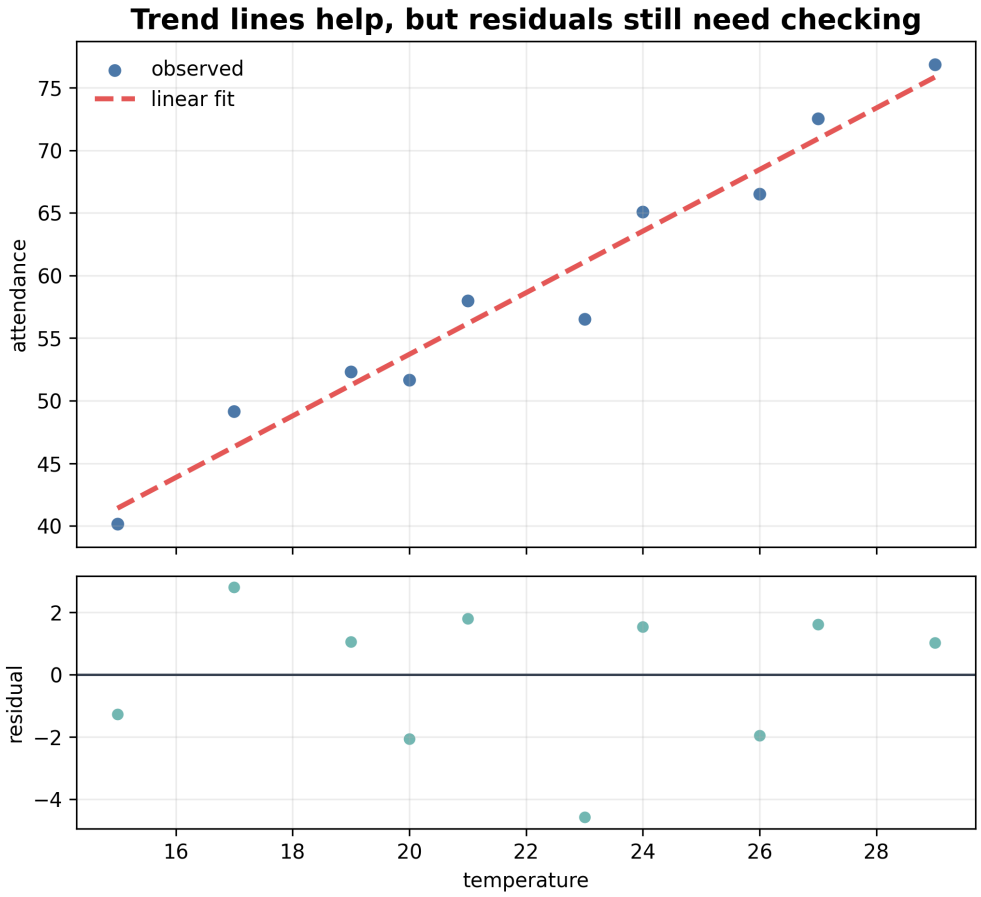


Figure 25.3: A trend line summarises the average relationship, while a residual panel shows whether the errors still contain structure that the line missed.

25.3.4 Interpretive caution: extrapolation and residual patterns

A trend line fitted between 15 and 24° remains reliable within that range but should not be extrapolated far beyond it. Predicting attendance at 50° assumes the linear relationship holds indefinitely, an assumption the data cannot support. Similarly, if residuals (the vertical gaps between observed points and the fitted line) form a funnel shape or systematic curve, the linear model fails to capture the underlying process. Use residual plots (predicted versus residual) to diagnose such patterns before presenting the trend line as definitive.

When overlaying multiple fits (for example, separate lines for weekday versus weekend attendance), ensure each line is labelled and coloured distinctly. Matplotlib legends (`ax.legend()`) become essential once the plot carries more than one trend, preventing readers from guessing which line corresponds to which subset.

Section summary

- Fit a linear model, extract slope and intercept, and confirm coefficients align with domain intuition.
- Build a two-row prediction DataFrame holding x-axis extremes, then call `lr.predict()` to generate y-endpoints.
- Draw the trend line with `ax.plot()`, styling it distinctly (dashed, coloured) to separate it from raw scatter points.
- Avoid extrapolating beyond the observed x-range and inspect residuals for non-linear patterns.

Self-check questions

1. Why do we create a prediction DataFrame with only two rows (minimum and maximum temperature) rather than predicting for every observed temperature?
   Hint: consider how Matplotlib draws a straight line from two endpoints.

2. Describe a scenario in which the scatter points lie far from the fitted line, and explain what that pattern suggests about model adequacy.
   Hint: think about residual spread and potential non-linearity.

3. If the regression line predicts negative attendance at very low temperatures, what does that reveal about the intercept term, and why might extrapolation be misleading?
   Hint: recall that attendance cannot fall below zero in reality.

25.4 Add reference lines and text annotations

Layer vertical or horizontal markers and plain-text labels to highlight thresholds, clusters, or outliers.

Learning objectives

After this section we will be able to:

- call `ax.axvline(x=...)` and `ax.axhline(y=...)` to draw reference boundaries;
- style reference lines with colour, line width, and dashed patterns so they remain visible without dominating the data;
- position text annotations with data coordinates to explain clusters or outliers;
- adjust line order (via plotting sequence) so reference markers sit behind scatter points and avoid obscuring data.

Prerequisites

Before proceeding we should:

- understand how to interpret x- and y-coordinates in data units (for example, sepal length in centimetres);
- recall earlier `axvline` and `axhline` syntax if we have already seen simple reference lines;
- recognise that plot layering follows the sequence of plotting commands.

Reference lines partition the data plane into regions—acceptable versus alert zones, training versus test ranges, or pre-intervention versus post-intervention periods. Text annotations label those regions or call attention to notable points. Together they transform a plain scatter or line plot into an annotated narrative that guides interpretation.

Example: threshold and label.

```
fig, ax = plt.subplots(figsize=(8, 6))

# Draw reference lines *before* scatter so they sit underneath
ax.axvline(x=6.0, color="lightgrey", linestyle="--", linewidth=0.5)
ax.axhline(y=5.0, color="darkgrey", linestyle="-.", linewidth=0.5)

ax.scatter(iris["sepal length (cm)"], iris["petal length (cm)"], c=colors, s=70,
↪  alpha=0.6)

ax.text(6.05, 6.5, "Really big plant", fontsize=10)
```

```
ax.set_xlabel("Sepal length (cm)")
ax.set_ylabel("Petal length (cm)")
ax.set_title("Our criteria for a big plant")
fig.tight_layout()
```

Readers see a vertical line at sepal length 6 cm and a horizontal line at petal length 5 cm, partitioning the plot into quadrants. The text "Really big plant" sits above and to the right of the intersection, labelling the region where both measurements exceed the thresholds. Because the lines are pale grey and dashed, they do not overpower the coloured markers.

Adjust `fontsize` to balance legibility with canvas real estate: too large and the text dominates; too small and readers squint. If the label overlaps data points, shift the `(x, y)` coordinates slightly or reduce marker `alpha` so the text remains readable. When layering multiple reference lines, vary line styles (`"-"`, `"-."`, `":"`) or colours so each boundary remains distinct.

Accessibility note. Reference lines and annotations should also appear in the figure caption or alt-text. A screen-reader user benefits from a summary such as: "Vertical dashed line at 6 cm sepal length and horizontal dash-dot line at 5 cm petal length partition the scatter into four regions. The upper-right quadrant is labelled 'Really big plant.'" That narration ensures the thresholds communicate even when colour and line style are invisible.

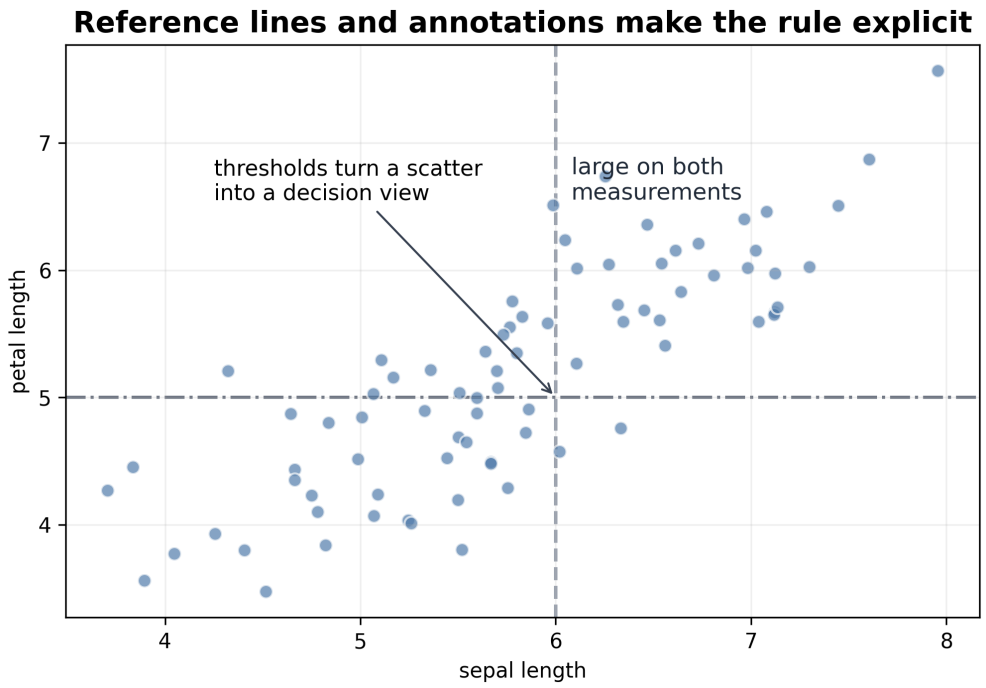


Figure 25.4: Reference lines and text annotations turn thresholds into an explicit visual rule. The same threshold information should be repeated in captions or alt-text.

Section summary

- Plot reference lines before scatter or bar data so they sit in the background layer.
- Use pale colours, dashed styles, and narrow line widths to avoid overwhelming the primary data.
- Position text annotations in data-coordinate units, iterating on placement to prevent overlap.
- Describe thresholds and labels in captions so accessibility tools convey the annotated regions.

Self-check questions

1. Why should reference lines be drawn before scatter points in the plotting sequence, and what visual problem arises if we reverse that order?
   Hint: consider how Matplotlib layers subsequent plot calls on top of earlier ones.

2. How would you adjust the text annotation coordinates if the label "Really big plant" overlaps a dense cluster of setosa markers?
   Hint: think about shifting x or y slightly or changing the anchor alignment.

Concluding bridge

This chapter combined intentional colour palettes, redundant marker shapes, bar charts for categorical data, regression trend lines, and annotated reference boundaries, and showed how to integrate the resulting figures into reports and dashboards. In chapter 2 we contrast these code-driven workflows with Tableau's drag-and-drop interface, clarifying when each tool best serves the data-communication goal.

Chapter exercises

1. Revise a colour-only scatter plot so that group membership is also visible through marker shape, annotation, or another redundant cue.
2. Fix a categorical bar chart whose y-axis makes a small difference look dramatic. Explain what changed.
3. Add a trend line to a scatter plot and write a caption that warns readers against over-interpreting it.
4. Add threshold lines and a text annotation to a plot. Write alt-text-style detail that communicates the same thresholds.

5. Critique a flawed figure and list the minimum edits needed before it belongs in a report.

Chapter 26

Build Reliable Pipelines in scikit-learn

Chapter overview

This is an optional computing-extension chapter. We now translate the tidy workflows we perfected in Orange into Python code. The chapter explains how scikit-learn pipelines help us keep feature engineering, preprocessing, imputation, and modelling in lock-step so data never drifts between steps. We use the road-crash workflow from chapter 23 as the main example, then bridge to the tic–tac–toe practical to show how the same pipeline can catch future-information leakage. The narrative builds on the guard rails in chapter 14 and prepares us to recreate the same discipline in a notebook.

26.1 Design column-aware pipelines before writing code

Plan transformations around the dataset schema so every column receives the right treatment and nothing slips through unprocessed.

Learning objectives

After working through this section, we will be able to:

- sketch a preprocessing plan that separates numeric, categorical, and derived features;
- choose the appropriate scikit-learn transformer for each feature type before opening a notebook;
- explain why pipelines reduce data leakage by bundling preprocessing with model training.

Prerequisites

Confirm that we can:

- read schema documentation or inspect a DataFrame to identify column types;
- interpret summary statistics to spot skew, missingness, and outliers;
- describe the purpose of the transformations we used in Orange (for example, normalisation, one-hot encoding, feature selection).

The scikit-learn pipeline API mirrors the Orange canvas metaphor: we draw the sequence before we click the buttons. Start by listing every column under headings such as numeric, categorical, target, metadata, and fields that must be dropped. Annotate the list with any bespoke steps such as extracting an hour from `Two-hour intervals`, parsing the number inside `Speed limit`, or removing severity columns that would leak the answer. Once the plan looks complete, translate it into a `ColumnTransformer` definition. A plain text table in your notebook or project notes is enough; the important part is the mapping, not a particular slide or screenshot.

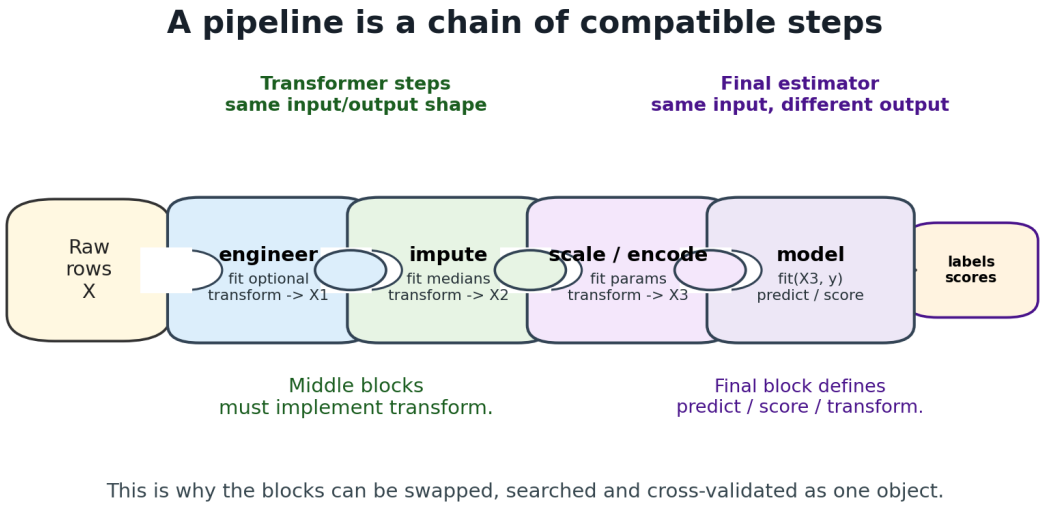


Figure 26.1: A pipeline is a chain of compatible steps. Preprocessing blocks transform the feature matrix, while the final estimator defines the prediction or score.

```
from sklearn.compose import (
    ColumnTransformer,
    make_column_selector as selector,
)
from sklearn.preprocessing import (
    OneHotEncoder,
    StandardScaler
)

injury_cols = [
    "No. killed",
```

```
    "No. seriously injured",
    "No. moderately injured",
    "No. minor-other injured",
]

drop_after_target = injury_cols + [
    "Degree of crash",
    "Degree of crash - detailed",
]

preprocess = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(),
         selector(dtype_include="number")),
        ("cat",
         OneHotEncoder(handle_unknown="ignore"),
         selector(dtype_exclude="number")),
    ],
    remainder="drop",
)
```

Document each choice in the project journal so teammates understand why certain fields were dropped or grouped. That transparency keeps the imputation logic defensible and makes leakage reviews much easier.

Accessibility spotlight. When drafting the column plan, share it in a text-based format (for example, a table in the project wiki) rather than an image so screen-reader users can review and edit the schema.

Section summary

- Treat the pipeline diagram as a design artefact before coding.
- Use `ColumnTransformer` to keep column mappings explicit and reproducible.
- Record transformation decisions so future readers can audit the workflow.

Self-check questions

1. What risks do we invite if we start coding without a column plan?
   Hint: Think about inconsistent scaling and dropped columns.
   Common misconception: Pipelines automatically infer the correct schema.
2. Why does setting `handle_unknown="ignore"` on `OneHotEncoder` matter in production?
   Hint: Consider new categories appearing after deployment.

Common misconception: The model will crash only when retrained, not during inference.

26.2 Select imputation strategies that match the data story

Choose imputers that respect the distribution of each feature and document when synthetic values enter the dataset.

Learning objectives

After this section you will be able to:

- distinguish between mean, median, constant, and learned imputations and justify each choice;
- encode missingness explicitly when it carries signal (for example, "unknown" categories or zero-inventory flags);
- audit the volume of imputed entries to set expectations with stakeholders.

Prerequisites

Before proceeding, ensure you can:

- interpret missing value heatmaps or percentage summaries;
- explain when median imputation outperforms the mean (for skewed numeric fields);
- describe how Orange handles missing values through its imputation tools and staged workflows.

Missing data is not a single problem. Sometimes the gap results from a genuine absence (for example, optional survey questions); other times it signals a broken integration. For numeric columns with symmetric distributions, a mean imputer keeps the totals balanced. Skewed variables, such as property prices, benefit from the median or from domain-aware defaults (e.g. replacing missing bedrooms with the mode). Categorical columns often deserve an explicit "Unknown" category so the model can learn whether absence correlates with outcomes.

Worked example: crash-injury risk

To see the planning mindset in action, continue with the crash workbook. Begin by splitting the schema into three lists:

- numeric signals such as `Distance`, `Latitude`, `Longitude`, `No. of traffic units involved`, and engineered numeric fields such as `hour` or `speed_limit`;

- categorical descriptors such as `Day of week of crash`, `Road surface`, `Weather`, `Natural lighting`, and `First impact type`;

- outcome or answer-like fields, including `No. killed`, `No. seriously injured`, `No. moderately injured`, `No. minor-other injured`, `Degree of crash`, and `Degree of crash - detailed`.

The outcome fields are useful for creating `y`, but they must not remain in `X`. This point is worth stressing: a model that sees injury counts while predicting whether an injury occurred is not predicting. It is reading the answer key.

We combine these ideas by adding imputers inside the column pipeline. Scikit-learn lets us chain steps within each branch so we never forget to scale after filling values.

```
from sklearn.compose import (
    ColumnTransformer,
    make_column_selector as selector,
)
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (
    OneHotEncoder,
    StandardScaler,
)

num_pipeline = Pipeline([
    ("impute", SimpleImputer(
        strategy="median")),
    ("scale", StandardScaler()),
])

cat_pipeline = Pipeline([
    ("impute", SimpleImputer(
        strategy="most_frequent")),
    ("encode", OneHotEncoder(
        handle_unknown="ignore")),
])

preprocess = ColumnTransformer(
    transformers=[
        ("num", num_pipeline,
         selector(dtype_include="number")),
        ("cat",
         cat_pipeline,
         selector(dtype_exclude="number")),
    ]
)
```

Strategy	Best suited for	Example rationale
Mean	Approximately symmetric data	Hospital bed-occupancy with stable seasonal averages.
Median	Skewed or heavy-tailed data	Property prices where extremes drag mean off centre.
Constant	Known safe defaults	Power-usage with "estimated consumption" flag.
Most frequent	Low-cardinality categoricals	Device platform; missing means common option.
Learned / model-based	Complex interactions	Triage where iterative imputation uses labs.

Table 26.1: Choosing an imputation strategy depends on both the distribution and the story you need to tell stakeholders. Use the table to explain to domain experts why you favoured a particular replacement.

After fitting, inspect the `SimpleImputer.statistics_` arrays to confirm the replacement values match expectations. Summarise the number of imputed records per feature in your project notes; stakeholders deserve to know when a large share of weather, lighting, or road-surface values relied on synthetic entries.

Try this variation. If missingness correlates strongly with the target, add a boolean "was_missing" feature through the `add_indicator=True` option on `SimpleImputer`. This mirrors the indicator columns we toggled in Orange and gives the downstream model an explicit flag to reason about.

Stakeholder conversation starter. Before finalising the imputers, share a quick summary with domain experts: "We filled missing numeric crash fields with medians and missing categorical fields with the most common category. Before we treat these scores as evidence, please check whether 'unknown' road surfaces or lighting conditions mean genuinely missing data or a meaningful reporting category." Conversations like this make synthetic data transparent and build trust with teams who rely on safety analysis.

Troubleshooting clinic.

- **Sparse/dense mismatches:** A downstream step that complains about receiving a sparse matrix (for example, an estimator or visualiser that only accepts dense arrays) usually means `OneHotEncoder` handed it a sparse result. Set `sparse_output=False` on `OneHotEncoder` so it returns a dense array, or wrap the step in a converter that densifies the input.
- **Unknown categories:** If production data introduces new payment methods, set `handle_unknown="ignore"` on `OneHotEncoder` so unseen categories are encoded as all-zero columns instead of raising an error. Refit and redeploy before the log fills with failed predictions.
- **Memory pressure:** When one-hot encoding creates a genuinely sparse matrix, prefer estimators that handle sparse input cleanly, such as logistic regression or linear SVM

variants. If you want histogram-based gradient boosting instead, switch to its native categorical workflow rather than assuming the one-hot pipeline will stay compatible. Profiling with `memory_profiler` helps catch transformers that silently densify arrays.

Advanced imputation options. For projects where the signal lives in subtle correlations, graduate beyond `SimpleImputer`. `IterativeImputer` cycles through regressors to estimate each missing column, while `KNNImputer` searches the nearest neighbours in feature space. Reserve these heavier tools for datasets with enough rows (think hospital records or credit-risk ledgers) and budget extra compute time for cross-validation.

Section summary

- Match imputation strategies to the distribution and business context of each column.
- Chain imputers and scalers inside pipelines so filled values flow through the same transformations as observed ones.
- Record replacement statistics and share them with the project team.

Self-check questions

1. Why might a constant value of zero be misleading when imputing missing stock counts?
   Hint: Consider the difference between "zero items remaining" and "information not recorded."
2. How would you convince a stakeholder that median imputation is safer than the mean for house prices?
   Hint: Reference the long right tail we observed in the land-value case study.

Progressive exercises

Tackle these in order to cement the workflow before the hands-on activity:

1. **Column audit:** Take a fresh dataset—for example, the open ICU mortality benchmark—and sketch the numeric/categorical/drop lists with one sentence explaining each decision.
2. **Imputer rationale:** For the same dataset, note which fields deserve median, constant, or learned imputers and justify them using Table 26.1.
3. **Pipeline evaluation:** Implement the plan in a notebook, add `add_indicator=True` to the most critical missing fields, and run `cross_validate`. Capture both accuracy and memory usage so you can report trade-offs to stakeholders.

26.3 Bundle preprocessing and modelling for leak-free evaluation

Train models through a single pipeline so cross-validation repeats every transformation consistently and we can deploy the exact same recipe.

Learning objectives

By the end of this section, you will be able to:

- wrap preprocessing and estimators in one `Pipeline` object for training and prediction;
- evaluate the pipeline with cross-validation and interpret variance across folds;
- export the fitted pipeline for deployment without re-running manual preprocessing steps.

Prerequisites

Make sure you can:

- explain the difference between training, validation, and test splits;
- compute accuracy, precision, recall, and RMSE depending on the task;
- use scikit-learn's `cross_val_score` or `cross_validate` helpers.

Wrapping everything into a pipeline means we call `fit` once and trust that scikit-learn handles the sequence. The crash workflow includes one custom feature-engineering step before the column-aware preprocessing: parse a numeric hour from `Two-hour intervals`, parse a numeric speed from `Speed limit`, and drop target-like severity columns before the estimator sees the data. The code below assumes `X` and `y` have been created as in chapter 23, with `y` indicating whether any injury occurred.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (
    FunctionTransformer,
)

def engineer(df):
    df = df.copy()
    df["hour"] = (
        df["Two-hour intervals"]
          .str.extract(r"^(\d{1,2})", expand=False)
          .astype(float)
          .fillna(-1)
```

```
    )
    df["speed_limit"] = (
        df["Speed limit"]
          .str.extract(r"(\d+)", expand=False)
          .astype(float)
          .fillna(-1)
    )
    return df.drop(columns=[
        "Two-hour intervals",
        "Speed limit",
        "Degree of crash",
        "Degree of crash - detailed",
    ], errors="ignore")

def make_crash_pipeline(estimator):
    transformer = FunctionTransformer(
        engineer, validate=False
    )
    return Pipeline([
        ("engineer", transformer),
        ("preprocess", preprocess),
        ("model", estimator),
    ])

log_clf = make_crash_pipeline(
    LogisticRegression(max_iter=1000)
)

scoring = {
    "accuracy": "accuracy",
    "roc_auc": "roc_auc",
}

cv_results = cross_validate(
    log_clf,
    X,
    y,
    scoring=scoring,
    n_jobs=-1,
)

print(
    cv_results["test_accuracy"].mean(),
    cv_results["test_roc_auc"].mean()
)
```

Because the pipeline owns the transformations, every fold sees only training data during imputation and scaling. That discipline guards against the accidental leakage that

chapter 14 warned us about. Review the mean and standard deviation for each metric to judge stability, and plot them over time as you tweak hyperparameters so the team can spot regressions quickly.

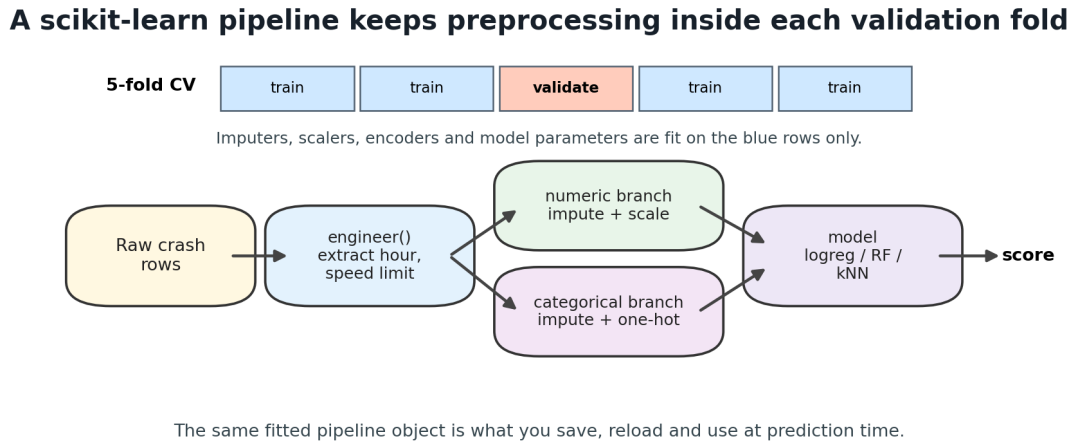


Figure 26.2: Cross-validation must refit preprocessing inside each training fold. Imputers, scalers, encoders, and model parameters are learned from the training rows only, then applied to the validation row.

If you want to use `HistGradientBoostingClassifier` instead, take a different route: keep the features in a DataFrame, mark categorical columns with categorical dtypes, and rely on the estimator's native categorical support rather than sparse one-hot encoding. Mixing the two stories in one example obscures the design choice.

Swap models without rewriting preprocessing

Once the feature-engineering and preprocessing steps are stable, model comparison becomes deliberately boring. Reuse the same wrapper and change only the final estimator:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier

rf_clf = make_crash_pipeline(
    RandomForestClassifier(random_state=42)
)
knn_clf = make_crash_pipeline(
    KNeighborsClassifier()
)

for name, model in [
    ("logistic regression", log_clf),
    ("random forest", rf_clf),
```

```
    ("k-nearest neighbours", knn_clf),
]:
    scores = cross_validate(
        model, X, y, scoring=scoring, n_jobs=-1
    )
    print(name, scores["test_roc_auc"].mean())
```

This is the Python equivalent of an Orange canvas where several learners feed into the same **Test & Score** widget. The fairness comes from sharing the same folds and the same preprocessing rules.

Tune named pipeline steps

Hyperparameter search uses the same pipeline object. The only new convention is the double underscore: `model__C` means "inside the step named `model`, tune the parameter called `C`." That naming pattern also works for earlier steps, such as `preprocess__num__impute__strategy`, though deeply nested grids quickly become hard to read.

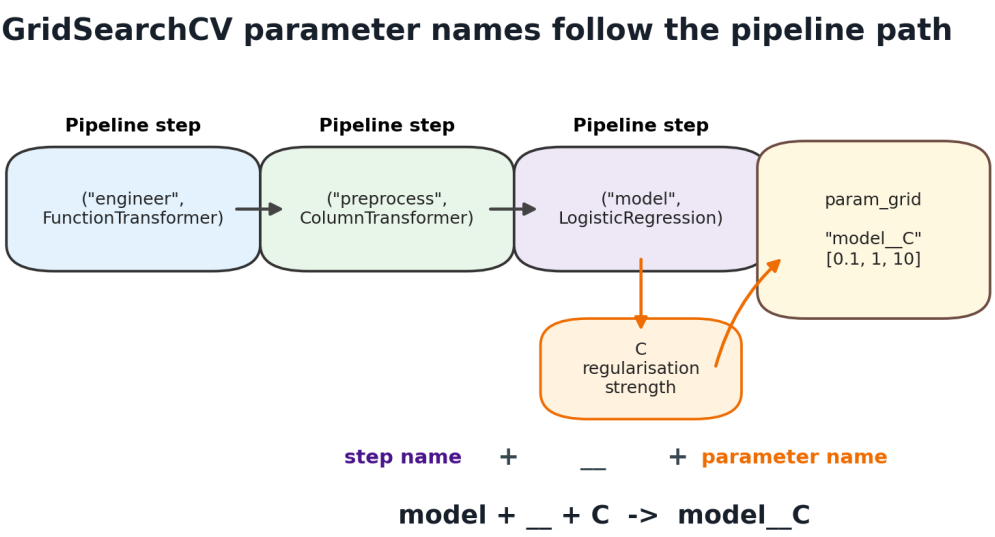


Figure 26.3: Pipeline parameter names follow the path through named steps. The double underscore lets `GridSearchCV` reach into the estimator at the end of the pipeline.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    "model__C": [0.1, 1, 10],
    "model__penalty": ["l2"],
    "model__solver": ["lbfgs"],
}
```

```
gs = GridSearchCV(
    log_clf,
    param_grid=param_grid,
    cv=5,
    scoring="roc_auc",
    n_jobs=-1,
)
gs.fit(X, y)
print(gs.best_params_, gs.best_score_)
best = gs.best_estimator_
```

Keep the grid small until the full pipeline runs reliably. A slow or broken grid search usually means the preprocessing step needs profiling before the model settings deserve attention.

Reporting tip. Log the exact pipeline object with `joblib.dump` and include the file hash in your project documentation. Reproducibility beats screenshots when auditors ask how the model was trained.

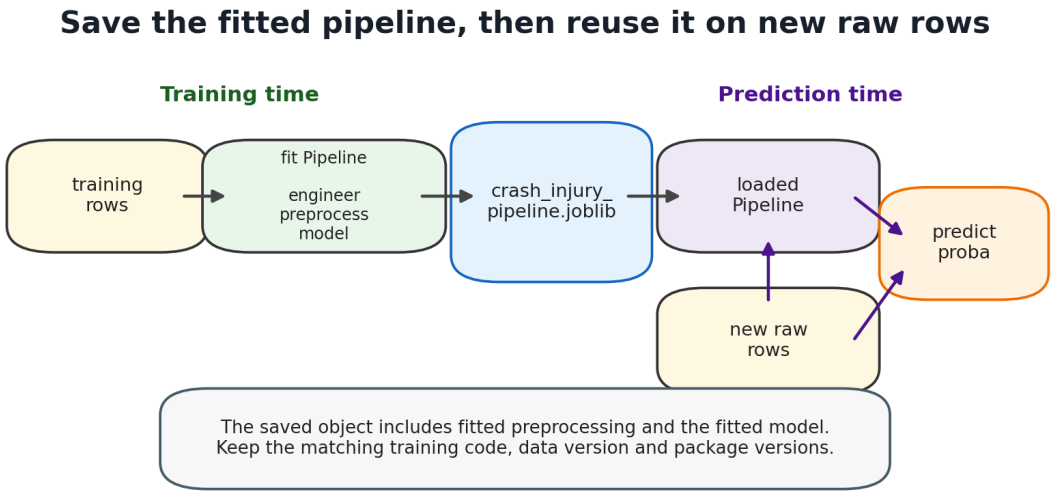


Figure 26.4: Training can be complicated while inference stays simple, provided the saved object includes the fitted preprocessing and the final estimator.

One practical warning matters here. `joblib.dump` stores the fitted pipeline state and a reference to custom Python functions, but it does not bundle source code from a notebook cell. If your pipeline contains `FunctionTransformer(engineer, validate=False)`, put `engineer` in an importable file such as `crash_pipeline.py` before relying on `joblib.load` in a later notebook or inference script. Also avoid loading model files from untrusted sources; unpickling is code execution, not a harmless data read.

Computational headroom. Pipelines are only as fast as their slowest transformer. Scaling wide one-hot matrices or training boosted trees on dense features can chew through gigabytes of RAM. Profile the preprocessing step with `memory_profiler` to spot bottlenecks, then consider:

- caching intermediate results with `Pipeline` using `memory=joblib.Memory(...)` when repeated cross-validation reuses the same imputations;
- keeping one-hot features sparse and choosing a sparse-friendly estimator when categorical spaces become very wide;
- chunking evaluation with `HalvingGridSearchCV` once you move to larger hyperparameter sweeps.

Document the runtime alongside metric tables so product managers understand the compute budget they are signing up for.

After fitting, inspect the transformed feature names and the model outputs together. For linear models, that might mean comparing the largest coefficients with the corresponding token or category names; for tree-based models, it might mean looking at feature-importance summaries. Either way, interpret the fitted pipeline as a whole rather than treating preprocessing and modelling as separate stories.

Section summary

- Pipelines align preprocessing and modelling so evaluation never reuses information improperly.
- Cross-validation on the pipeline yields honest performance estimates and highlights unstable folds.
- Serialising the fitted pipeline simplifies deployment and future audits.

Self-check questions

1. What does it mean if one cross-validation fold scores dramatically lower than the others?
   Hint: Investigate class imbalance or unhandled categories in that split.
2. Why do we prefer exporting the entire pipeline over saving individual transformers?
   Hint: Think about guaranteeing the same preprocessing logic during inference.
3. Why does a custom `FunctionTransformer` need extra care when saving a fitted pipeline?
   Hint: Think about where Python will find the function when the model is loaded later.

26.4 Practical bridge: tic–tac–toe without future moves

Use a tiny game dataset to make leakage visible before returning to larger crash-data pipelines.

The hands-on activity uses tic–tac–toe because the prediction time is easy to see. After move 3 we know `move1`, `move2`, and `move3`; we do not yet know `move4` through `move9`. If a model uses all nine moves to predict the final result, it may score well, but it is no longer answering the honest question.

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (
    StratifiedKFold,
    cross_val_score,
)
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder

data = pd.read_csv("all_tictactoe_games.csv")
y = data["result"]
X = data[["move1", "move2", "move3"]]
cv = StratifiedKFold(
    n_splits=3, shuffle=True, random_state=0
)

dummy = DummyClassifier()
print(cross_val_score(dummy, X, y, cv=cv).mean())
print(cross_val_score(
    dummy, X, y, cv=cv, scoring="f1_macro"
).mean())

def make_ttt_pipeline(columns):
    return Pipeline([
        ("prep", ColumnTransformer([
            ("moves",
             OneHotEncoder(
                 handle_unknown="ignore"
             ),
             columns)
        ])),
        ("model",
         LogisticRegression(max_iter=500)),
    ])

pipe = make_ttt_pipeline(X.columns)
```

```
print(cross_val_score(
    pipe, X, y, cv=cv, scoring="f1_macro"
).mean())
```

Now run the same pipeline with all nine move columns. The comparison is a leakage audit, not a competition:

```
all_moves = [f"move{i}" for i in range(1, 10)]
X_all = data[all_moves]
pipe_all = make_ttt_pipeline(X_all.columns)
print(cross_val_score(
    pipe_all, X_all, y, cv=cv, scoring="f1_macro"
).mean())
```

If the all-move model jumps sharply, ask whether the extra columns would really be available when the prediction is made. This compact example is the same idea as dropping `No. killed` from the crash features: future information and outcome information both create impressive but dishonest scores.

After fitting the honest three-move model, inspect the transformed feature names and the per-class coefficient rows together. For a multiclass logistic regression, `coef_` has one row per class, so pair each row with `classifier.classes_` instead of assuming `coef_[0]` tells the whole story. Permutation importance gives a complementary view by shuffling `move1`, `move2`, and `move3` one at a time and measuring the score drop.

26.5 Where to next

The pipeline discipline here underpins the bag-of-words text classifier in the next extension chapter and the larger hyperparameter sweeps that follow. Keep the column plan, imputation log, and serialised pipeline nearby so you can extend the workflow into feature unions or model stacks without redoing the groundwork.

Chapter exercises

1. Sort the fields in a supervised dataset into numeric features, categorical features, metadata, target, and dropped columns.

2. Choose imputation strategies for at least five fields. Justify mean, median, mode, constant, indicator, or learned imputation choices using the data story.

3. Implement a `ColumnTransformer` plus estimator inside a scikit-learn pipeline, including a small feature-engineering step if the raw fields need parsing.

4. Identify whether a preprocessing step is inside or outside cross-validation. Rewrite the unsafe version so the transformation is fitted only on training folds.

5. Explain why `model__C` is the correct parameter name when tuning the logistic-regression step in a pipeline whose final step is named `model`.

6. In the tic–tac–toe practical, compare a model using only `move1`–`move3` with a model using all nine moves. Explain which version answers the honest prediction question.

7. Describe the fitted pipeline artefact, importable helper code, and metadata needed to reuse it safely on new data.

Chapter 27

Text as Data: From Documents to Vectors to Visualisations

Chapter overview

Data science no longer confines itself to numbers and tables; text, email, speech transcripts, and social media now feed models just as readily as spreadsheet columns. This chapter introduces two complementary approaches: large language models that interpret context at scale, and bag-of-words representations that transform documents into interpretable feature vectors. We focus on the latter—showing how `CountVectorizer` turns sentences into sparse arrays, how n-grams capture local context, how TF–IDF and regularisation refine a classifier, and how dimensionality reduction through UMAP lets us visualise thousands of messages in two dimensions. The chapter connects these foundations to spam classification, coefficient auditing, and exploratory text visualisation.

27.1 Choose between large language models and bag-of-words

Recognise when interpretability and compute efficiency favour traditional vectorisation over neural approaches.

Learning objectives

After this section we will be able to:

- describe the two dominant paradigms for analysing text at scale;
- articulate when large language models excel (automated labelling, nuanced classification) and when bag-of-words suffices (interpretable reporting, lightweight exploration);
- explain why interpretability matters when stakeholders need to understand which words drive predictions.

Prerequisites

We should already be comfortable with:

- the concept of supervised machine learning (features, labels, train/test splits);
- basic familiarity with ChatGPT or similar conversational AI tools;
- interpreting logistic regression coefficients as indicators of feature importance.

Modern text analysis offers two broad options. **The first option** is large language models accessed via API—tools such as ChatGPT, Claude, or domain-specific fine-tuned transformers. These models excel at tasks requiring contextual understanding. Examples include labelling product descriptions into taxonomy categories, triaging customer support tickets by urgency, and extracting structured data from unstructured narratives. The cost and latency vary by provider. Even so, automating annotation for hundreds of thousands of documents becomes feasible where manual labelling would require months of human effort.

The second option is bag-of-words representations paired with explainable classifiers such as logistic regression or decision trees. This approach treats each document as a collection of word counts, ignoring sentence structure but preserving frequency patterns. The resulting models train quickly on modest hardware and scale to millions of documents without GPU clusters. They also produce coefficients that reveal which terms most strongly predict each outcome. That interpretability matters when stakeholders ask, "What words signal that a customer intends to close their account?" or "Which phrases in call-centre transcripts correlate with escalated complaints?" The trade-off is that bag-of-words models sacrifice nuance. They cannot distinguish "Australia beat New Zealand" from "New Zealand beat Australia" because both sentences contain the same set of words.

For many applied text projects, the classification score is only the first check. The useful output is often the language evidence: which words or phrases separate one class from another, whether those signals make domain sense, and whether the model is leaning on shortcuts that would embarrass us in front of a stakeholder. Bag-of-words keeps that audit trail visible.

Consider a customer-feedback project where a service provider already knew which customers had left poor ratings. The useful question was not "who is unhappy?" but "what are they unhappy about?" A bag-of-words model made the answer visible: comments mentioning bathrooms and changing rooms were strongly associated with low scores. That result led to an operational fix—cleaning and staffing decisions—rather than another round of abstract sentiment modelling. The model mattered because its coefficients pointed back to a plain-language problem the organisation could act on.

Choose large language models when the task demands contextual reasoning, when budget permits API costs, and when post-hoc explainability matters less than raw accuracy. Choose bag-of-words when interpretability guides business decisions, when compute resources are constrained, or when the dataset is large enough that millions of documents need lightweight feature extraction. Often the best workflow combines both. Use an LLM to bootstrap

labels for a small sample, then train a bag-of-words classifier on those labels. That scales annotation across the full corpus while retaining auditability.

Professional context. Bag-of-words methods dominated AI for decades and remain competitive and highly interpretable for many practical tasks because they are cheap, fast, and transparent. Understanding both paradigms ensures we select the right tool for the deadline, the stakeholder, and the available infrastructure.

Section summary

- Large language models provide contextual understanding at API cost; bag-of-words offers interpretability at compute efficiency.
- Interpretability becomes essential when stakeholders demand transparency about which features drive predictions.
- Hybrid workflows leverage LLMs for labelling and bag-of-words for scalable, explainable classification.

Self-check questions

1. Under what conditions would you recommend a bag-of-words model over a large language model API for a client's text classification project?
   Hint: consider budget, latency, interpretability requirements, and dataset scale.

2. How does the inability to distinguish word order affect bag-of-words performance on sentiment analysis tasks?
   Hint: think about negation ("not good") and how n-grams partially address the limitation.

27.2 Transform text into sparse count matrices

Use `CountVectorizer` to convert documents into feature vectors while managing vocabulary size and memory.

Learning objectives

After this section we will be able to:

- explain the bag-of-words mental model: each dimension corresponds to a vocabulary word, and each document becomes a row of counts;
- instantiate `CountVectorizer` from scikit-learn and call `fit_transform` on a list of strings;

- distinguish between sparse and dense arrays, and recognise when sparse storage prevents memory exhaustion;
- retrieve the learned vocabulary to audit which terms the model recognises.

Prerequisites

Make sure we can:

- load text data into a pandas Series or Python list;
- recall how `fit_transform` workflows operate from scaling and clustering contexts;
- interpret NumPy arrays and understand zero-based indexing.

The bag-of-words model starts with a corpus—a collection of documents, where each document might be a sentence, a paragraph, an email, or an entire article. `CountVectorizer` scans the corpus to build a vocabulary of all unique words (or tokens), sorts them alphabetically, and assigns each word to a column index. Every document then becomes a row of counts: the value at column j records how many times word j appears in that document.

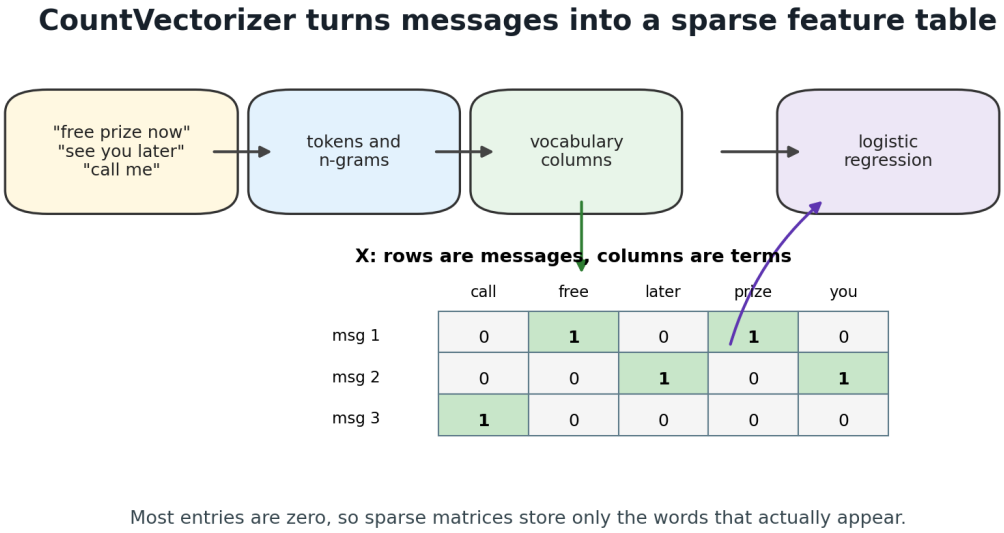


Figure 27.1: `CountVectorizer` turns raw messages into a sparse table whose rows are documents and whose columns are tokens or n-grams. A classifier can then learn from those numeric columns.

Consider a minimal example with four short sentences:

Example: basic count vectorisation.

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
```

```
sentences = pd.Series([
    "the quick brown fox",
    "the slow brown fox",
    "roses are red",
    "the red red fox",
])

cvec = CountVectorizer()
sparse_array = cvec.fit_transform(sentences)
print(sparse_array)
# Output: (4, 8) sparse matrix with 4 rows (documents) and 8 columns (vocabulary)

print(cvec.get_feature_names_out())
# ['are', 'brown', 'fox', 'quick', 'red', 'roses', 'slow', 'the']
```

The output shows a sparse matrix with shape $(4, 8)$: four documents, eight unique words. Converting to a dense array reveals the counts:

```
print(sparse_array.toarray())
# [[0 1 1 1 0 0 0 1]   # "the quick brown fox"
#  [0 1 1 0 0 0 1 1]   # "the slow brown fox"
#  [1 0 0 0 1 1 0 0]   # "roses are red"
#  [0 0 1 0 2 0 0 1]]  # "the red red fox" (note: 'red' appears twice)
```

Each row corresponds to one sentence. The first sentence contains "brown" (column 1), "fox" (column 2), "quick" (column 3), and "the" (column 7), yielding ones in those positions and zeros elsewhere. The fourth sentence includes "red" twice, so column 4 holds a count of 2.

Sparse versus dense storage. When the vocabulary grows to tens of thousands of words and the corpus scales to millions of documents, most entries in the count matrix remain zero—any single document uses only a small fraction of the total vocabulary. Sparse storage records only the nonzero entries (row index, column index, count value), dramatically reducing memory usage. A dense array for 100,000 documents with 50,000 features would consume approximately 20 GB of RAM at 4 bytes per float; the equivalent sparse representation might fit in 200 MB if each document averages 100 unique tokens. Convert to dense only when necessary for small datasets or debugging; otherwise keep the sparse format for pipeline compatibility and memory efficiency.

Vocabulary auditing. Export the learned vocabulary with `cvec.get_feature_names_out()` and inspect it for unexpected tokens. Typographical errors, HTML tags, or numeric codes may inflate the vocabulary without adding predictive value. Configure `CountVectorizer` with `lowercase=True`, `stop_words="english"`, or `max_features=5000` to prune noise and keep the feature space interpretable.

Tokenisation defaults are choices. The default token pattern in scikit-learn treats words as runs of letters or numbers with at least two characters. That means one-letter words such as "I" vanish, punctuation is ignored, and an apostrophe can split `don't` into `don` while dropping the one-letter `t`. This may be harmless in an SMS spam model, where `don eat` can only plausibly come from `don't eat`. It may be misleading in a customer database with many people named Don. If the default vocabulary looks wrong, set an explicit `token_pattern`, adjust preprocessing, or use a tokeniser suited to the language and domain.

Rare does not automatically mean useful. A token that appears in one or two documents may be a meaningful product code or a specialised phrase, but it may also be a typo, a tracking number, a person's name, or a copy-and-paste artefact. Treat rare-token pruning as a judgement call: inspect examples before choosing `min_df`, and keep domain phrases only when they help the model generalise rather than memorise accidents in the training set.

`max_features` introduces another judgement call. It keeps the most frequent features, but the final slot can be arbitrary when many tokens have the same count. A word that appears once is sometimes a meaningful singleton, but it is often just a spelling mistake or a phrase that will never appear again at prediction time. Use frequency caps to keep the model readable, then audit the boundary cases instead of assuming the cap made a clean semantic choice.

Section summary

- `CountVectorizer` builds a sorted vocabulary from the corpus and represents each document as a row of word counts.
- Sparse matrices store only nonzero entries, preventing memory exhaustion on large text datasets.
- Retrieve the vocabulary to audit tokens and guide preprocessing decisions.

Self-check questions

1. How would the count matrix change if you added the sentence "brown fox brown fox" to the corpus?
   Hint: consider which columns increment and by how much.

2. Why might a sparse matrix be essential for a corpus with 50,000 vocabulary words but only 100 unique tokens per document on average?
   Hint: calculate the proportion of zero entries and their memory footprint.

27.3 Capture local context with n-grams

Expand the feature space to include word pairs and triplets so the model distinguishes "not good" from "good".

Learning objectives

After this section we will be able to:

- define unigrams (single words), bigrams (word pairs), and trigrams (word triplets);
- configure `CountVectorizer` with `ngram_range=(1, 2)` to extract both unigrams and bigrams;
- explain how n-grams partially mitigate the word-order limitation of bag-of-words models;
- recognise the vocabulary explosion that occurs when increasing the n-gram range.

Prerequisites

Before proceeding we should:

- understand basic `CountVectorizer` usage from the previous section;
- appreciate that word order conveys meaning ("not happy" differs from "happy");
- be prepared to manage larger feature spaces through `max_features` or `min_df` pruning.

Bag-of-words ignores word order, treating "my cat is good" and "my cat is not good" identically because both contain the same tokens. N-grams restore some context by treating consecutive word sequences as distinct features. A **bigram** captures every pair of adjacent words: "my cat", "cat is", "is good". A **trigram** extends to three-word sequences. By including both unigrams and bigrams in the vocabulary, the model learns that "not good" carries a different signal than "good" alone.

Configuring `ngram_range=(1, 2)` instructs `CountVectorizer` to extract all unigrams and all bigrams from the corpus:

Example: unigrams and bigrams.

```
cvec_bigram = CountVectorizer(ngram_range=(1, 2))
sparse_array_bigram = cvec_bigram.fit_transform(sentences)

print(cvec_bigram.get_feature_names_out())
# ['are', 'are red', 'brown', 'brown fox', 'fox', 'quick',
#  'quick brown', 'red', 'red fox', 'red red', 'roses',
#  'roses are', 'slow', 'slow brown', 'the', 'the quick',
#  'the red', 'the slow']
```

The vocabulary now includes single words ("brown", "fox") and pairs ("brown fox", "quick brown"). The fourth sentence, "the red red fox", generates the bigrams "the red", "red red", and "red fox". Sparse storage remains essential: adding bigrams typically multiplies the vocabulary size by a factor of five to ten, yet most documents still use only a small fraction of the expanded feature space.

Trade-offs. N-grams improve context sensitivity at the cost of vocabulary explosion. A corpus with 10,000 unigrams might yield 80,000 bigrams and 300,000 trigrams. Constrain growth with `max_features` to cap the vocabulary at interpretable sizes, or use `min_df=5` to discard n-grams that appear in fewer than five documents after checking that this does not remove meaningful rare phrases. Monitor training time and memory usage; if the pipeline slows unacceptably, reduce `ngram_range` or prune features more aggressively.

When to stop. For most classification tasks, `ngram_range=(1, 2)` strikes a balance between context and tractability. Trigrams help domain-specific phrase matching ("intensive care unit", "customer service representative") but demand careful pruning. Evaluate whether the added complexity improves cross-validation scores before committing to higher n-gram ranges.

Section summary

- N-grams capture local word order by treating consecutive token sequences as distinct features.
- Configure `ngram_range=(1, 2)` to include both unigrams and bigrams in the vocabulary.
- Vocabulary size grows rapidly with n-gram range; use `max_features` and `min_df` to manage memory and interpretability.

Self-check questions

1. How would you explain the difference between "not good" and "good" to a stakeholder reviewing bag-of-words coefficients?
   Hint: highlight how the bigram "not good" receives its own weight separate from "good".

2. When might you choose `ngram_range=(1, 3)` over `ngram_range=(1, 2)`, and what computational trade-off would you accept?
   Hint: consider domain-specific multi-word phrases and the corresponding vocabulary expansion.

27.4 Audit a text classifier with coefficients

Use cross-validation and coefficient inspection to check whether a fast text model has learned a sensible pattern.

Learning objectives

After this section we will be able to:

- build a minimal text-classification pipeline with a vectoriser and logistic regression;
- evaluate the model against a simple baseline before interpreting its coefficients;
- explain how TF–IDF and regularisation change the meaning and stability of the learned weights.

Prerequisites

We should already be comfortable with:

- cross-validation from chapter 14;
- logistic regression coefficients as feature weights;
- the vocabulary and n-gram controls introduced earlier in this chapter.

The SMS spam dataset is a useful running example because the labels are simple and the language evidence is easy to inspect. Each row contains a short message labelled either legitimate or spam. A practical workflow is:

1. build a pipeline that vectorises the text and fits a logistic regression model;
2. estimate performance with cross-validation;
3. compare that score with a dummy baseline that always predicts the majority class;
4. refit on the full training corpus only after the validation check looks sane;
5. extract the vocabulary and coefficients to see which tokens drove the model.

Example: count-vector spam classifier.

```
import pandas as pd
from sklearn.feature_extraction.text import (
    CountVectorizer
)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (
    cross_val_score
)
```

```
from sklearn.pipeline import make_pipeline

model = make_pipeline(
    CountVectorizer(ngram_range=(1, 2)),
    LogisticRegression()
)

scores = cross_val_score(
    model, messages, labels, cv=5
)
print(scores.mean())

model.fit(messages, labels)
steps = model.named_steps
vectorizer = steps["countvectorizer"]
classifier = steps["logisticregression"]
coefficients = pd.Series(
    classifier.coef_[0],
    index=vectorizer.get_feature_names_out()
)
```

If spam makes up only a small fraction of the corpus, a classifier can appear respectable by predicting "not spam" nearly every time. That is why the dummy baseline matters. Once the real model clears that low bar, inspect precision and recall as well as accuracy. In a junk-mail setting, a false positive can hide a legitimate message in the spam folder. That is usually more harmful than letting one spam message through. The metric should reflect the cost of the mistake, not just the percentage of correct labels.

Coefficient inspection turns the model from a score into an explanation. Positive coefficients push messages towards the positive class, while negative coefficients push them towards the negative class. In a spam model, words such as `free`, `reply`, or `txt` should look plausible as spam indicators; personal pronouns or conversational words may point towards legitimate messages. If the largest coefficients are phone numbers, row identifiers, label words, or one-off artefacts, the model may be exploiting leakage rather than learning a reusable language pattern.

These coefficients should be visualised, not just printed. A horizontal bar chart of the most positive and most negative terms usually communicates the result better than a long table. A word cloud can be a quick presentation aid, provided the caption explains that word size represents coefficient strength rather than raw frequency. A plot of coefficient against document frequency is useful for audit work because it separates rare but strong tokens from common but weak ones.

TF–IDF weighting. Raw counts reward frequent words. TF–IDF combines term frequency with inverse document frequency, downweighting words that appear almost everywhere and emphasising terms that are locally important in a smaller subset of documents. It may or may not improve accuracy on a particular corpus; its real value is

often interpretive. If a common word is still strongly predictive after TF–IDF, the model is telling us that the word matters despite appearing widely.

Regularisation. Text classifiers often have more columns than rows, especially after adding bigrams. Logistic regression can still work, but it needs pressure not to chase accidental phrases with huge coefficients. Regularisation adds that pressure: the model is asked to predict well while also keeping the coefficients small. `LogisticRegressionCV` can choose the regularisation strength by cross-validation, giving a principled way to balance fit and stability before reporting the most influential tokens.

Beyond spam. The same workflow applies whenever the words themselves are evidence. A historical-text project might label passages as mythic or historical, train a bag-of-words classifier, and then inspect which words push the model towards each label. Terms connected with family, monuments, materials, or money can become signals worth discussing with domain experts. The point is not that the coefficient is the interpretation by itself; the coefficient gives us a ranked shortlist of language patterns to investigate.

Section summary

- Cross-validation checks whether a text classifier generalises before we interpret its coefficients.
- Coefficients turn a bag-of-words classifier into an auditable language model, but they must be checked for leakage and artefacts.
- TF–IDF changes the weighting of common terms, while regularisation keeps coefficients from chasing accidental phrases.

Self-check questions

1. Why is a dummy baseline useful before celebrating a high spam-classifier accuracy?
   Hint: consider class imbalance and the majority class.

2. A token has a very large coefficient but appears in only one training document. What checks would you run before trusting it?
   Hint: think about leakage, typos, and whether the word can recur in future data.

27.5 Understand the curse of dimensionality

Recognise why high-dimensional text features challenge visualisation and intuition, motivating dimensionality reduction techniques.

Learning objectives

After this section we will be able to:

- articulate why a 10,000-dimensional feature space defies human spatial reasoning;
- explain how most points in high-dimensional space sit far from the centre, counterintuitively clustering near the "surface";
- motivate dimensionality reduction as a necessary preprocessing step for exploratory visualisation.

Prerequisites

We should already be comfortable with:

- interpreting scatter plots in two or three dimensions;
- basic probability concepts (uniform distribution, expected value);
- recognising that `CountVectorizer` can produce thousands of features.

A vocabulary of 10,000 words creates a 10,000-dimensional space where each document becomes a point. Human intuition fails at such scales: we cannot sketch or mentally rotate a 10,000-dimensional cube. Even at 18 dimensions—the feature count after vectorising the four-sentence toy corpus with bigrams—geometric properties diverge from our 2D or 3D experience.

Consider randomly placing 1,000 points inside an 18-dimensional unit hypercube. In two dimensions, we would expect some points near the centre $(0.5, 0.5)$ and others scattered toward the edges. In 18 dimensions, almost every point sits far from the centre, with most clustering near the hypercube's surface. The median distance from the origin grows with dimensionality, meaning "close" and "far" lose their everyday interpretations. This **curse of dimensionality** manifests in several ways:

- **Sparsity.** Most pairwise distances become similar, making nearest-neighbour searches unreliable without massive datasets.
- **Visualisation paralysis.** We cannot plot 10,000 dimensions on a screen; any attempt to inspect the data directly fails.
- **Interpretability strain.** Stakeholders struggle to grasp which of thousands of features drive model predictions.

Bag-of-words amplifies the curse: adding bigrams or trigrams pushes dimensionality into tens of thousands, and sparse storage only addresses memory—not the conceptual barrier. The practical remedy is **dimensionality reduction**: projecting the high-dimensional data onto two or three dimensions while preserving as much structure as possible. That projection enables exploratory scatter plots, cluster identification, and sanity checks before committing to supervised learning pipelines.

Why not just ignore it? Machine learning algorithms often handle high-dimensional data competently—logistic regression with L2 regularisation, for instance, remains effective even when features outnumber samples. The curse matters most when humans need to *see* the data: identifying outliers, spotting annotation errors, or explaining to stakeholders how spam messages cluster separately from legitimate ones. Dimensionality reduction bridges the gap between algorithmic scalability and human interpretability.

Section summary

- High-dimensional feature spaces defy geometric intuition; points cluster near boundaries rather than centres.
- The curse of dimensionality makes exploratory visualisation and outlier detection impossible without projection.
- Dimensionality reduction techniques compress thousands of features into 2D or 3D coordinates that humans can interpret.

Self-check questions

1. How does the curse of dimensionality affect nearest-neighbour classifiers when vocabulary size reaches 50,000 features?
   Hint: consider how distances converge and what that means for identifying "similar" documents.

2. Why might a data scientist prioritise dimensionality reduction for an initial exploratory analysis even if the final model will use all 10,000 features?
   Hint: think about stakeholder communication and early pattern detection.

27.6 Visualise high-dimensional text with UMAP

Apply UMAP to project bag-of-words features onto a 2D scatter plot that reveals document clusters and outliers.

Learning objectives

After this section we will be able to:

- explain UMAP's goal: preserve local neighbourhood structure when projecting from high to low dimensions;
- install `umap-learn` and run it after vectorisation to produce 2D embeddings;

- interpret a UMAP scatter plot where spam and legitimate messages form visually distinct clusters;
- recognise that UMAP cannot perfectly preserve all distances; it approximates the manifold structure.

Prerequisites

Make sure we can:

- run `pip install umap-learn` or `conda install umap-learn` in a Python environment;
- create scatter plots with Matplotlib;
- understand that `fit_transform` workflows stack: vectoriser → UMAP → 2D coordinates.

UMAP (Uniform Manifold Approximation and Projection) reduces dimensionality by modelling the high-dimensional data as a manifold—a curved surface embedded in many dimensions—and then finding a low-dimensional layout that approximates the same neighbourhood relationships. The algorithm proceeds in three conceptual steps:

1. **Identify neighbours.** For each document, determine which other documents sit closest in the original high-dimensional space. UMAP assigns a probability that any two points "belong together" based on distance and a tunable `n_neighbors` parameter (commonly 15–50).
2. **Project to low dimensions.** Initialise 2D coordinates (UMAP uses a spectral embedding by default) for every document and compute the same neighbour-probability structure in the low-dimensional embedding.
3. **Nudge iteratively.** Adjust the 2D coordinates to make the low-dimensional probabilities match the high-dimensional probabilities as closely as possible. The optimisation cannot achieve a perfect match—compressing 10,000 dimensions into two inevitably loses information—but it approximates the manifold well enough for visualisation.

The code workflow mirrors `CountVectorizer`: fit the vectoriser on the corpus, transform documents into sparse counts, then pass those counts to a UMAP model and fit-transform again to obtain 2D coordinates:

Example: UMAP projection of SMS spam dataset.

```
from sklearn.feature_extraction.text import CountVectorizer
from umap import UMAP
import matplotlib.pyplot as plt
import pandas as pd
```

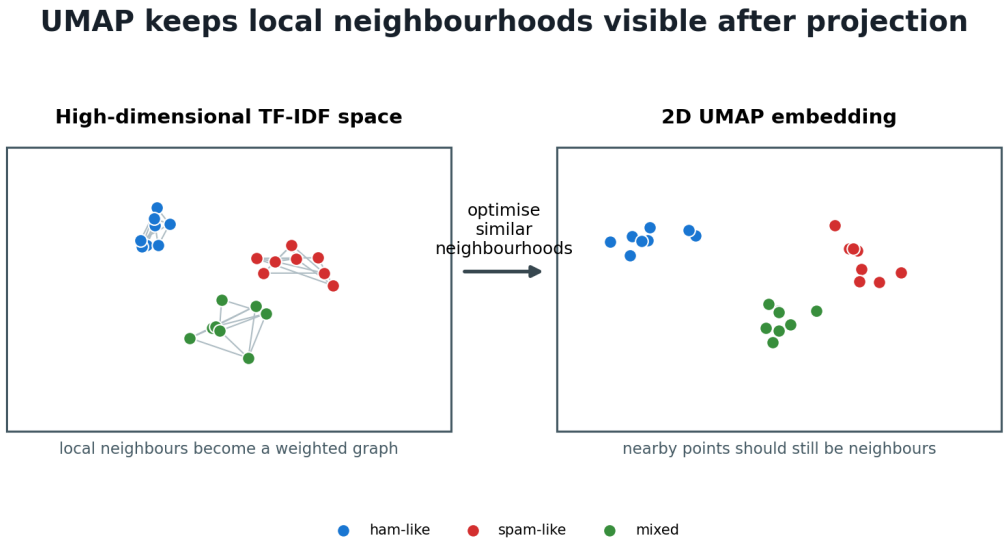


Figure 27.2: UMAP tries to preserve local neighbourhoods when projecting high-dimensional text features into two dimensions. The low-dimensional plot is useful for exploration, but it does not preserve every original distance.

```
# Load SMS spam dataset (labels: "spam" or "ham")
sms_data = pd.read_csv("sms_spam_collection.csv")

# Vectorise messages
cvec = CountVectorizer(ngram_range=(1, 2), max_features=5000)
X_sparse = cvec.fit_transform(sms_data["message"])

# Reduce to 2D
umap_model = UMAP(n_neighbors=30, min_dist=0.1, random_state=42)
X_2d = umap_model.fit_transform(X_sparse)

# Plot
fig, ax = plt.subplots(figsize=(8, 6))
spam_mask = sms_data["label"] == "spam"
ax.scatter(
    X_2d[~spam_mask, 0], X_2d[~spam_mask, 1],
    c="steelblue", s=20, alpha=0.6, label="Ham (legitimate)"
)
ax.scatter(
    X_2d[spam_mask, 0], X_2d[spam_mask, 1],
    c="crimson", s=20, alpha=0.7, label="Spam"
)
ax.set(xlabel="UMAP dimension 1", ylabel="UMAP dimension 2",
       title="SMS messages projected to 2D via UMAP")
ax.legend()
```

```
fig.tight_layout()
```

For a quick teaching plot, fitting UMAP on the full corpus is acceptable. In a modelling workflow, split the data first. Fit the vectoriser and UMAP only on the training messages, and use `transform` for held-out messages if you need to place them on the same axes. Do not use the test set to choose vocabulary settings, UMAP parameters, or model hyperparameters; reserve it for final confirmation.

The resulting scatter plot reveals that spam messages (red) cluster together in one region, while legitimate messages (blue) dominate another. Some spam points scatter among the legitimate cloud—UMAP cannot separate every ambiguous case—but the broad structure confirms that word-usage patterns differ systematically between the two classes. This visualisation serves as a sanity check before training a classifier: if UMAP shows complete overlap, the vocabulary might need pruning or the labels might contain errors.

Tuning UMAP. The `n_neighbors` parameter controls how many nearby points influence each document's position. Smaller values (10–15) emphasise local structure and produce tighter clusters; larger values (50–100) reveal global topology at the cost of merging fine details. The `min_dist` parameter sets the minimum spacing between points in the embedding; lower values (0.0–0.1) pack clusters tightly, while higher values (0.3–0.5) spread them apart for easier annotation. Experiment with both to balance clarity and fidelity to the high-dimensional structure.

Limitations. UMAP optimises for preserving neighbourhoods, not exact distances. Two documents that appear adjacent in the 2D plot might still differ substantially in the original 10,000-dimensional space. Use UMAP for exploration and stakeholder communication, not as ground truth for similarity measurements. When precise distance matters, rely on the original sparse matrix and cosine similarity or Euclidean norms.

Accessibility spotlight. Describe UMAP scatter plots with alt-text that summarises the cluster count, their separation, and any outliers. For example: "Two-dimensional UMAP projection showing approximately 4,500 legitimate SMS messages (blue) forming a diffuse cloud in the centre-right, with 500 spam messages (red) clustering tightly in the upper-left quadrant; a handful of red points scatter among the blue, indicating ambiguous cases."

Section summary

- UMAP projects high-dimensional bag-of-words features onto 2D or 3D coordinates by approximating neighbourhood structure.
- Fit the vectoriser first, then pass the sparse matrix to `UMAP().fit_transform()` to obtain plottable coordinates.
- Interpret the resulting scatter plot as an exploratory tool; it reveals clusters and outliers but does not preserve exact distances.

Self-check questions

1. How would increasing `n_neighbors` from 15 to 100 change the appearance of a UMAP scatter plot?
   Hint: consider whether the algorithm emphasises local tight clusters or global spread.

2. Why might a UMAP projection show two distinct clusters even when a logistic regression classifier achieves only 80% accuracy?
   Hint: think about what UMAP optimises (neighbourhood preservation) versus what the classifier optimises (decision boundary).

Concluding bridge

Text data becomes analysable once we convert words into counts. This chapter introduced the bag-of-words paradigm through `CountVectorizer`, showed how n-grams capture local context, and explained how coefficient auditing turns a classifier into evidence we can discuss. We also saw why the curse of dimensionality motivates visualisation tools such as UMAP, which project thousands of features onto two dimensions so we can spot clusters and outliers.

These same techniques apply cleanly to the SMS spam dataset: building a count vectoriser, experimenting with unigrams and bigrams, training a logistic regression classifier, and visualising the corpus with UMAP before and after filtering stop words. The next chapter (chapter 28) revisits the same pipeline, layering on hyperparameter search and cross-validation to tune vocabulary size, n-gram ranges, and regularisation strengths systematically. Together, these tools transform raw text into interpretable, auditable models that scale from hundreds to millions of documents without requiring GPU clusters or API budgets.

Text-model terminology map. The names in this chapter are easy to blur together. Keep the following map in mind when reading code or explaining a result.

- **Text to numbers.** A token is the word-like unit we count. An n-gram is a short sequence of tokens. A vocabulary is the list of tokens or n-grams that become columns. `CountVectorizer` builds that vocabulary and turns documents into a sparse matrix.
- **Model evidence.** A logistic regression model learns a weight for each text column. A large positive or negative coefficient points to language associated with a class. Cross-validation checks whether the pattern survives across different training folds. Precision and recall help us decide whether the model is making the kind of mistake stakeholders care about.
- **Refinements and views.** Document frequency counts how many documents contain a term. TF-IDF downweights terms that appear almost everywhere. Dimensionality

is the number of feature columns. UMAP gives us a two-dimensional view of that high-dimensional table, useful for exploration but not proof by itself.

The short version: text becomes a big sparse table; the model learns weights; the weights point us back to the words worth inspecting.

Chapter exercises

1. For several text-analysis tasks, decide whether a bag-of-words model or a large language model is the better starting point. Explain the constraint that matters most.

2. Fit or inspect a `CountVectorizer` vocabulary. Identify two tokens to remove, two to keep, and one decision that needs domain judgement.

3. Compare unigram and bigram features on a small SMS sample. Which bigrams add meaning that single words miss?

4. Inspect the top positive and negative coefficients from a spam classifier. Which terms look like useful evidence, and which look like possible leakage or data artefacts?

5. Explain why a sparse text matrix is difficult to inspect directly, even when the original messages are short.

6. Interpret a UMAP projection cautiously. Name one pattern worth following up and one claim the plot cannot prove.

Chapter 28

Tune Text Models with Hyperparameter Search

Chapter overview

This chapter shows how hyperparameter search keeps models honest while a count vectoriser turns raw text into features we can tune. We explain how cross-validation underpins any search routine, then build a reusable pipeline that marries tokenisation, vocabulary management, and linear models. The closing section turns that strategy into a concrete notebook workflow so readers can compare settings and document their findings.

28.1 Treat hyperparameter search as a structured experiment

Plan search grids around defensible questions and evaluate each candidate with cross-validation rather than a single lucky split.

Learning objectives

After this section we will be able to:

- describe how cross-validation estimates generalisation performance for each candidate configuration;
- design a search space that balances model flexibility with computation time;
- explain the guard rails that prevent leakage between the validation fold and the test set.

Prerequisites

Confirm that we can:

- interpret cross-validation score summaries (mean, standard deviation, minimum);

- trace how scikit-learn pipelines bundle preprocessing and modelling work from chapter 26;
- document experimental settings in a project journal or issue tracker.

Hyperparameters control how expressive a model may become before it memorises training quirks. Grid search is the simplest approach: define a dictionary of candidate values, evaluate each with cross-validation, and keep the configuration that delivers the best validation score without inflating variance. Randomised search samples the space instead of enumerating it, which is practical when the grid would otherwise explode combinatorially.

Regardless of the method, treat the search like any other experiment. Start by naming the question: are we exploring regularisation strengths, n-gram widths, or both? Sketch a table that lists each hyperparameter, the rationale for its range, and any computational limits. For example, you might narrow the logistic regression penalty to a short list of inverse regularisation strengths (C values) so the first pass stays interpretable and affordable.

Once the grid is locked, specify a scorer that matches the business objective. Accuracy suffices for balanced sentiment datasets; F_1 or area under the ROC curve better reflect imbalanced complaint logs. When results return, chart both the mean score and its spread. A configuration that wins by a fraction of a percent but doubles variance rarely improves the deployed workflow. Keep the untouched test set for a single, final confirmation run after the search concludes.

Stakeholder check-in. Share the proposed search grid with collaborators before running it. Doing so avoids situations where a teammate expects character 5-grams while you only trial unigrams and bigrams, and it surfaces runtime constraints that might dictate a random search instead of an exhaustive grid.

Section summary

- Formulate the search question first, then encode it as a grid or sampling distribution.
- Cross-validate each candidate so the reported score reflects unseen data, not a single split.
- Present both the mean and spread of validation scores to guide the final choice.

Self-check questions

1. How does increasing the number of cross-validation folds influence the stability of your search results?
   Hint: balance statistical reliability against compute time.
   Common misconception: more folds always improve accuracy without side-effects.

2. When might a randomised search outperform a grid search even on a small dataset?
   Hint: consider interactions across multiple hyperparameters.
   Common misconception: grids always cover the "important" combinations.

28.2 Vectorise text data with reproducible preprocessing

Transform raw documents into structured features while keeping the vocabulary manageable and explainable.

Learning objectives

After this section you will be able to:

- outline the stages of a bag-of-words pipeline (tokenisation, normalisation, vocabulary restriction);
- configure scikit-learn's `CountVectorizer` to match project requirements;
- integrate text features into a pipeline that feeds a linear classifier alongside tabular context.

Prerequisites

Before diving in, ensure you can:

- explain basic natural language concepts such as tokens, stop words, and n-grams;
- load and inspect text datasets in pandas;
- interpret logistic regression coefficients.

The bag-of-words model counts how often each token appears in a document, ignoring order but preserving frequency patterns. We start with `CountVectorizer` because it supports reproducibility well: fixing the random state and vocabulary keeps the pipeline deterministic when we rerun it or share it with teammates.

Configure the vectoriser with a clear story about your audience. Customer-support tickets might prioritise lowercasing and moderate stop-word removal, while triage notes from a community health clinic may demand a wider n-gram window to capture symptom phrases and abbreviations. In both cases, cap the maximum feature count so the model stays interpretable for reviewers. Pair the vectoriser with a `TfidfTransformer` if relative term importance outweighs raw counts. Keep an eye on memory usage: each additional n-gram expands the sparse matrix, so restrict `max_features` or `max_df` to prune ubiquitous filler words.

Be careful with rare terms. A rare token can be the exact phrase that matters in a specialist setting, but it can also be a typo, a user ID, a once-off proper name, or leakage

from the labelling process. Inspect examples before increasing `min_df`, and explain whether the choice is about speed, noise reduction, privacy, or generalisation.

When combining text with tabular features, wrap the vectoriser inside a pipeline branch and feed it into a `ColumnTransformer` alongside numeric preprocessing. The scaffold below illustrates a minimal setup for classifying triage notes while retaining vital-sign summaries from chapter 26.

```
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import (
    CountVectorizer
)
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

text_features = "triage_note"
numeric_features = [
    "heart_rate",
    "respiratory_rate",
    "acuity_score"
]

model = Pipeline([
    ("features", ColumnTransformer([
        ("text", CountVectorizer(
            lowercase=True,       # normalise
            ngram_range=(1, 2),   # phrases
            max_features=5000,    # cap
            min_df=5,             # very rare terms
        ), text_features),
        ("numeric", Pipeline([
            ("scale", StandardScaler()),
        ]), numeric_features),
    ])),
    ("clf", LogisticRegression()),
])
```

Save the fitted vocabulary in version control. In scikit-learn, `get_feature_names_out()` exports that list for you, which makes later audits much easier.

Accessibility spotlight. Provide alternate text summaries for any word clouds or heatmaps derived from the vectorised features so readers who rely on screen readers receive the same analytical insight.

Section summary

- Align vectoriser options with the communication needs of your audience and the constraints of the dataset.
- Combine text and tabular preprocessing through `ColumnTransformer` to keep the workflow reproducible.
- Version the learned vocabulary to support audits and fairness reviews.

Self-check questions

1. Why might you increase `min_df` before raising `max_features`?
   Hint: think about rare tokens that add noise to the classifier.
   Common misconception: more features always improve accuracy.
2. How could you adapt the pipeline when new jargon enters the ticket queue over time?
   Hint: consider scheduled retraining and vocabulary diffing.
   Common misconception: vectorisers automatically absorb unseen words without refitting.

28.3 Coordinate search routines with your notebook workflow

Translate the design decisions above into concrete workflow steps.

Set up a notebook that uses `GridSearchCV` over the count vectoriser and logistic regression hyperparameters. Prepare by sketching three candidate grids:

- an "express" grid for a quick run with two C values and unigram-only features that finishes in roughly 30–45 seconds on an 8-core laptop;
- a "balanced" grid for the main exercise, trialling unigrams and bigrams alongside three C values that completes in about 3–4 minutes on the same hardware;
- an "exploratory" grid for independent work that tests adding trigram features or swapping in a linear support vector machine, which can stretch to 8–10 minutes on shared machines.

Set expectations early by listing the runtime budget and recommended hardware (a quad-core CPU with 8 GB of RAM suffices for the first two grids) so readers can choose the right tier. Annotate the notebook with markdown cells that remind you when to downgrade the search if the environment slows down. Close with a comparison table that lists the best parameters, validation scores, and the top weighted tokens. That summary creates a clean bridge into the recommendation chapter, where we revisit ranking metrics such as precision@k and mean reciprocal rank alongside personalised text features.

Beyond grids. Mention Bayesian optimisation libraries such as Optuna or Hyperopt for curious readers. These tools model the objective surface and adaptively sample promising regions, which can trim compute budgets once we outgrow fixed grids.

Connection to practice. Log every search run, even "failed" ones, in a shared project journal. Hyperparameter work is iterative; leaving a paper trail saves teammates from repeating an unhelpful configuration.

Chapter exercises

1. Design an express search grid for vectoriser and logistic-regression parameters that should finish during a short run.
2. Interpret a `GridSearchCV` result table. Choose a model using score, variance, and runtime rather than score alone.
3. Compare lowercase, stop-word, unigram, and bigram settings. Explain which trade-off each setting makes.
4. Fill a search-run log recording seed, folds, parameters, runtime, metric, and selected model.
5. Choose one extra search dimension worth adding next and one that would be wasteful under the current time budget.

Chapter 29

Network Analysis: Graphs, Centrality, and Shortest Paths

Chapter overview

Most data science work treats rows as independent observations—predicting whether transaction 13 is fraudulent requires no information about transaction 12 or 14. Yet many real-world systems exhibit structure where entities connect to each other: people form friendships, web pages link to related content, road intersections channel traffic flows. Network analysis models these connections explicitly, revealing which nodes sit at the centre of influence, how quickly information spreads, and which paths minimise travel time. This chapter introduces NetworkX, Python's graph-manipulation library. With it we construct networks, visualise them with physics-inspired layouts, compute shortest paths using Dijkstra's algorithm, compare centrality measures, and find communities. We conclude by showing how these ideas transfer into social-network analytics and recommendation systems.

29.1 Recognise when network structure matters

Identify problems where relationships between entities drive the question more than individual attributes.

Learning objectives

After this section we will be able to:

- articulate scenarios where ignoring network structure produces misleading predictions;
- distinguish between standalone network analysis (e.g., identifying influencers) and feature engineering (e.g., centrality as a column in a supervised model);
- explain how network metrics supplement traditional tabular machine learning.

Prerequisites

We should already be comfortable with:

- supervised machine learning workflows (features, labels, train/test splits);
- interpreting data frames where each row represents an independent observation;
- recognising that some business questions require understanding relationships rather than isolated attributes.

Until now, every dataset we have analysed fits the spreadsheet paradigm: rows of independent observations, columns of features, and a target variable to predict. A retailer's transaction log treats each purchase as standalone; whether customer 42 bought milk does not inherently affect whether customer 43 bought bread. Supervised machine learning thrives in this setting.

Many problems violate that independence assumption. Traffic at the corner of two major roads depends on upstream congestion at nearby intersections: if a bottleneck diverts cars through alternative routes, flow patterns shift throughout the network. Fraud detection benefits from knowing whether a vendor has transacted with other accounts flagged as suspicious—a merchant connected to multiple confirmed fraud cases raises a red flag even if the current transaction looks benign in isolation. Social-media persuasion depends on cascades: a user's opinion shifts when friends share content, those friends influence their own networks, and the effect propagates through interconnected communities.

Network analysis serves two complementary purposes:

1. **Standalone exploration.** Identify the most influential nodes in a social network, rank them by centrality, and allocate advertising budgets to those key influencers. The network itself answers the business question without feeding a downstream classifier.
2. **Feature engineering.** Compute centrality scores, clustering coefficients, or shortest-path distances and add them as columns in a data frame. Train a supervised model predicting happiness from centrality, or churn risk from a customer's position in a referral network. The network-derived features augment traditional tabular learning.

Choosing between these approaches depends on the stakeholder's question. If the goal is "Find the top ten influencers," standalone network metrics suffice. If the goal is "Predict which customers will churn," centrality becomes one feature among many in a logistic regression or random forest model.

Professional context. Industries with inherently networked data—telecommunications (call graphs), logistics (route planning), finance (transaction networks), social media (user connections)—maintain dedicated graph databases and visualisation pipelines. Network terminology is shared vocabulary on these specialised analytics teams.

Section summary

- Network analysis applies when relationships between entities drive the question more than individual attributes.
- Use standalone network metrics to rank nodes (e.g., influencers); use derived features to augment supervised models.
- Common applications include traffic routing, fraud detection, social influence, and recommendation systems.

Self-check questions

1. How would you explain to a stakeholder when adding network features to a churn model is worth the engineering effort?
   Hint: consider whether customer-to-customer referrals or shared-vendor patterns exist in the data.

2. When might a pure network analysis (no supervised model) be the final deliverable rather than an intermediate step?
   Hint: think about questions that ask "who are the key nodes?" versus "will this node churn?"

29.2 Build graphs with NetworkX

Instantiate nodes and edges programmatically to represent relationships in data.

Learning objectives

After this section we will be able to:

- import NetworkX and create an empty graph object;
- add edges with `G.add_edge(node1, node2)`, understanding that nodes are created implicitly;
- query a node's degree (the count of its connections) using `G.degree[node]`;
- recognise that node identifiers can be strings, integers, or any hashable Python type.

Prerequisites

Make sure we can:

- install packages with `pip install networkx` or `conda install networkx`;
- recall object-oriented concepts (constructors, methods, attributes) from prior programming experience;
- interpret dictionaries and loops in Python.

NetworkX treats graphs as collections of **nodes** (also called vertices) connected by **edges** (also called links). Start by importing the library and instantiating an empty graph:

```
import networkx as nx

G = nx.Graph()
```

The variable `G` now holds a graph object with no nodes and no edges. Add edges by calling `G.add_edge(node1, node2)`; NetworkX automatically creates any referenced nodes that do not already exist. For example, connecting actors to TV shows:
Example: building an actor–show network.

```
G.add_edge("Friends", "Jennifer Aniston")
G.add_edge("Friends", "Courteney Cox")
G.add_edge("Friends", "Matthew Perry")
G.add_edge("The Morning Show", "Jennifer Aniston")
G.add_edge("The Morning Show", "Reese Witherspoon")
```

After these calls, `G` contains six nodes: three actors, two shows, and edges linking actors to the shows they appear in. Node names are arbitrary—strings, integers, tuples—as long as they are hashable and unique.

Query a node's **degree**, the count of edges connected to it, with `G.degree[node_name]`:

```
print(G.degree["Jennifer Aniston"])  # Output: 2 (connected to Friends and The
↪  Morning Show)
print(G.degree["Friends"])           # Output: 3 (connected to three actors)
```

Degree serves as a simple centrality measure: nodes with high degree occupy central positions, while nodes with degree 1 sit at the periphery. More sophisticated centrality metrics exist, but degree provides an intuitive starting point.

Directed and weighted graphs. The examples above use `nx.Graph()`, which creates an *undirected* graph: if A connects to B, B automatically connects to A. Real-world networks sometimes require *directed* edges: a one-way street, a Twitter follow relationship (Alice

follows Bob, but Bob does not follow Alice), or a citation (paper A cites paper B). Construct a directed graph with `nx.DiGraph()`.

Edges can also carry **weights** representing strength (frequency of interaction), cost (road distance), or capacity (bandwidth). Add weights by passing a third argument: `G.add_edge(nodeA, nodeB, weight=5)`. Weighted shortest-path algorithms (Dijkstra's) leverage these weights to find optimal routes. For introductory exploration, unweighted graphs suffice.

Data ingestion. Manually calling `add_edge` for hundreds of relationships becomes tedious. In practice, loop over a pandas DataFrame or a database query result:

```
import pandas as pd

edges_df = pd.read_csv("actor_show_edges.csv")  # columns: actor, show
for _, row in edges_df.iterrows():
    G.add_edge(row["actor"], row["show"])
```

This pattern scales to millions of edges when the data resides in a relational database or a graph-specific store such as Neo4j.

From a physical model to a graph. The same idea can start on the desk before it moves into Python. We can write member names on slips of paper, write each group code on a separate slip, and use short pieces of thread to connect each member to the groups they belong to. The result is a **bipartite graph**: member nodes connect to group nodes, but members do not connect directly to other members. A member-to-member path therefore alternates through group codes:

```
Alex -> COMP101 -> Harper
```

That physical setup maps directly to a NetworkX graph built from a spreadsheet:

```
unit_cols = ["Unit1", "Unit2", "Unit3", "Unit4"]
G_classes = nx.Graph()

for unit in unit_cols:
    for name, code in zip(class_df["Name"], class_df[unit]):
        if pd.notna(code):
            G_classes.add_edge(name, code)
```

The slips are nodes and the thread pieces are edges. Once the graph exists, shortest paths, centrality, and resilience questions become ordinary graph queries: find the path between two members, identify the most connected group, then remove that node and see whether the network splits.

Section summary

- Import NetworkX with `import networkx as nx` and create an empty graph with `G = nx.Graph()`.
- Add edges via `G.add_edge(node1, node2)`; nodes are created implicitly if they do not exist.
- Query node degree with `G.degree[node]` to count connections.
- Load edges from pandas DataFrames, shared spreadsheets, or database cursors to populate graphs programmatically.
- A bipartite membership network connects members to groups; member-to-member paths travel through group nodes.

Self-check questions

1. How would you modify the actor–show example to represent a directed graph where edges point from actors to the shows they appear in?
   Hint: replace `nx.Graph()` with `nx.DiGraph()` and consider edge directionality.

2. Why does NetworkX allow any hashable type as a node identifier, and what practical advantage does that offer over forcing integer IDs?
   Hint: think about code readability and avoiding a separate node-lookup table.

29.3 Visualise networks with spring layouts

Apply physics-inspired algorithms to position nodes on a 2D canvas, revealing clusters and central hubs.

Learning objectives

After this section we will be able to:

- explain the spring-layout metaphor: edges act as springs that pull connected nodes together;
- compute a layout with `pos = nx.spring_layout(G)` and interpret the returned dictionary mapping nodes to (x, y) coordinates;
- render the graph with `nx.draw_networkx(G, pos)` and customise figure size for readability;
- recognise layout limitations: 2D projections distort edge lengths and cannot preserve all distances.

Prerequisites

We should already know how to:

- create and populate a NetworkX graph as demonstrated in the previous section;
- produce Matplotlib figures with `plt.subplots` and set `figsize` for readability;
- understand that visualisations serve exploratory purposes rather than exact geometric truth.

A graph with ten nodes and fifteen edges has no inherent spatial layout. Visualising it on a 2D screen requires choosing (x, y) coordinates for each node so that connected nodes sit near each other and the overall structure remains legible. The **spring layout** algorithm treats edges as springs: imagine connecting every pair of linked nodes with a stretchy coil, then releasing the network and letting physics settle it into equilibrium.

Edges that stretch long exert tension pulling nodes together; nodes surrounded by many neighbours cannot move far without stretching multiple springs. The algorithm iterates until forces balance, producing a layout where tightly connected clusters bunch together and peripheral nodes drift toward the edges.

Compute the layout with:

```
pos = nx.spring_layout(G)
```

The variable `pos` holds a dictionary mapping each node to a tuple `(x, y)`. For example:

```
print(pos["Friends"])           # (0.23, -0.41)
print(pos["Jennifer Aniston"])  # (0.31, -0.38)
```

Pass `pos` to `nx.draw_networkx` to render the graph:

Example: visualising an actor network.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(12, 8))
nx.draw_networkx(G, pos, ax=ax)
ax.set_title("Actor--Show Network")
fig.tight_layout()
```

The resulting plot displays nodes as circles (or dots) labelled with their identifiers, connected by lines representing edges. TV shows that share multiple actors cluster together; isolated shows drift to the periphery. Adjusting `figsize` enlarges the canvas so labels do not overlap.

Layout limitations. No 2D projection can perfectly preserve distances from a graph with arbitrary connectivity. The spring layout prioritises local structure (keeping connected nodes close) over global fidelity. Edge lengths in the drawing do not represent weights or true graph distances; they reflect the layout algorithm's compromise between clarity and physics. Treat visualisations as exploratory tools that reveal clusters and central hubs, not as quantitatively accurate maps.

Interactive visualisations. For presentations or web dashboards, the D3 JavaScript library enables interactive network diagrams where users drag nodes, click to expand neighbourhoods, and watch springs animate in real time. Learning D3 sits beyond the scope of this chapter, but awareness that such tools exist helps when stakeholders request dynamic, browser-based network explorers. A common division of labour pairs NetworkX for analysis with D3 for presentation.

Section summary

- Spring layouts simulate physics by treating edges as springs that pull connected nodes together.
- Compute layouts with `nx.spring_layout(G)` and render with `nx.draw_networkx(G, pos)`.
- Layouts prioritise clarity and cluster visibility over exact distance preservation.
- Adjust figure size to prevent label overlap; consider D3 for interactive dashboards.

Self-check questions

1. Why does the spring-layout algorithm produce different node positions each time it runs, even on the same graph?
   Hint: consider random initialization and iterative optimization.

2. When might a circular layout or hierarchical layout be more appropriate than a spring layout?
   Hint: think about organisational charts or directed acyclic graphs.

29.4 Compute shortest paths with Dijkstra's algorithm

Find optimal routes through weighted or unweighted networks to minimise travel cost or hop count.

Learning objectives

After this section we will be able to:

- articulate the shortest-path problem: given a source and a target, find the path with minimum total edge weight;
- explain Dijkstra's greedy strategy: iteratively visit the nearest unvisited node and update distances;
- call `nx.shortest_path(G, source, target)` to retrieve the path as a list of nodes;
- recognise applications: GPS routing, supply-chain logistics, social-network degrees of separation.

Prerequisites

Before proceeding we should:

- understand graph fundamentals (nodes, edges, weights) from the previous sections;
- be comfortable with greedy algorithms conceptually (making locally optimal choices);
- recognise that "shortest" can mean fewest edges (unweighted) or minimum total weight (weighted).

The **shortest-path problem** asks: given a graph, a source node, and a target node, find the path that minimises total cost. In an unweighted graph, cost equals hop count; in a weighted graph, cost sums the edge weights along the path. Applications span logistics (minimise delivery time), navigation (minimise driving distance), and social networks (count degrees of separation between two people).

Dijkstra's algorithm solves this problem greedily:

1. Initialize all nodes with distance ∞ except the source (distance 0). Mark all nodes unvisited.
2. Identify the unvisited node with the smallest known distance. Visit it.
3. For each neighbour of the visited node, calculate the distance from the source via the current node. If that distance is smaller than the neighbour's current known distance, update it.
4. Repeat until the target is visited or all reachable nodes are exhausted.

Example: weighted shortest path.

Consider a graph with nodes A, B, C, D, E, F and weighted edges. We want the shortest path from A to E:

```
Initial:   A=0, all others=inf, unvisited={A, B, C, D, E, F}
Step 1:    Visit A (smallest=0). Update neighbors: C=2, B=4.
Step 2:    Visit C (smallest=2). Update neighbors: B=min(4,3)=3, D=10, E=12.
Step 3:    Visit B (smallest=3). Update neighbors: D=min(10,8)=8, F=8.
Step 4:    Visit D (smallest=8). Update neighbors: E=min(12,10)=10,
↪  F=min(8,11)=8.
Step 5:    Visit F (smallest=8). No updates.
Step 6:    Visit E (smallest=10). Done.
```

The algorithm confirms the shortest path A → C → B → D → E with total cost 10.

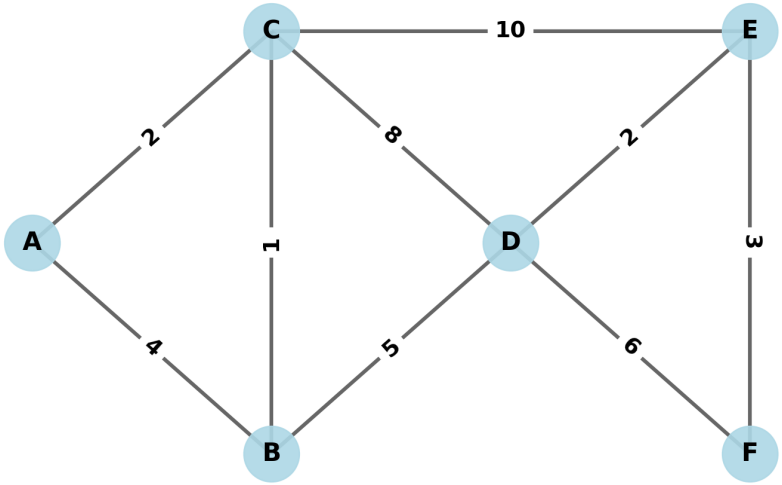


Figure 29.1: A weighted shortest-path example. Dijkstra's algorithm repeatedly visits the unvisited node with the lowest known distance and updates neighbouring distances.

NetworkX provides `nx.shortest_path`, which runs Dijkstra internally:

```
path = nx.shortest_path(G, source="A", target="E", weight="weight")
print(path)  # ['A', 'C', 'B', 'D', 'E']
```

Omitting the `weight` argument treats the graph as unweighted, counting hops instead of summing weights.

Six Degrees of Kevin Bacon. A famous application asks: can any two actors be connected through shared movies in six steps or fewer? Build a graph where actors are nodes and edges link actors who appeared in the same film. Shortest-path queries answer "How many degrees separate Actor X from Kevin Bacon?"

```
path = nx.shortest_path(actor_graph, source="Jennifer Aniston", target="Kevin
↪  Bacon")
print(len(path) - 1)  # Number of edges (degrees of separation)
```

If no path exists—two actors belong to disconnected components—NetworkX raises `NetworkXNoPath`.

Complexity and alternatives. Dijkstra's algorithm runs in $O((V + E) \log V)$ time with a priority queue, where V is node count and E is edge count. For very large graphs, specialised techniques such as A* (heuristic-guided search) or bidirectional Dijkstra (search from both ends) reduce computation by exploring fewer nodes before reaching the target.

Section summary

- Shortest paths minimise total edge weight (or hop count in unweighted graphs).
- Dijkstra's algorithm greedily visits the nearest unvisited node and updates neighbour distances.
- Call `nx.shortest_path(G, source, target)` to retrieve the optimal route.
- Applications include GPS navigation, logistics, and social-network degrees of separation.

Self-check questions

1. How would Dijkstra's algorithm behave if all edge weights were equal to 1?
   Hint: relate it to breadth-first search and hop counting.

2. Why does NetworkX raise an exception when no path exists, and how should you handle that in production code?
   Hint: consider try-except blocks and reporting disconnected components to stakeholders.

29.5 Compare centrality measures and network resilience

Turn graph structure into node-level scores, then ask how the network changes when important nodes disappear.

Learning objectives

After this section we will be able to:

- distinguish degree, closeness, betweenness, eigenvector/PageRank, and current-flow centrality;
- choose a centrality measure that matches the stakeholder's question;
- compute centrality scores in NetworkX and store them as data-frame features;
- explain how targeted node removal tests network resilience.

Prerequisites

We should already understand:

- nodes, edges, and connected components;
- shortest paths and average path length;
- data frames as feature tables for later modelling.

Centrality asks which nodes matter most, but "matter" changes with the problem. The most connected node is not always the best bridge, and the best bridge is not always the node closest to everything else. A transport planner, a social-media analyst, and a fraud team may each need a different definition of importance.

- **Degree centrality** rewards direct connections. In an airline network, it highlights airports with many direct routes.
- **Closeness centrality** rewards short paths to the rest of the network. It is useful when the question is how quickly a node can reach everyone else.
- **Betweenness centrality** rewards brokerage. A node scores highly when many shortest paths pass through it.
- **Eigenvector centrality** and **PageRank** reward connection to already-important nodes. These measures are useful when influence compounds through neighbours.
- **Current-flow centrality** imagines flow spreading through all available paths, not only the single shortest path.

The practical choice is domain-driven. For a direct-mail campaign, degree centrality may identify people with the most contacts. For a product-seeding campaign, eigenvector centrality may be more useful: a person with fewer direct followers but strong ties to other influential people can be a better route into the network. In a logistics network, betweenness can identify refuelling points, transfer stations, or bottlenecks that many routes must cross.

NetworkX exposes these scores as dictionaries keyed by node. We can combine them into a data frame for inspection or downstream modelling:

```
import pandas as pd

centrality_df = pd.DataFrame({
    "degree": nx.degree_centrality(G),
    "closeness": nx.closeness_centrality(G),
    "betweenness": nx.betweenness_centrality(G),
    "pagerank": nx.pagerank(G),
}).rename_axis("node").reset_index()
```

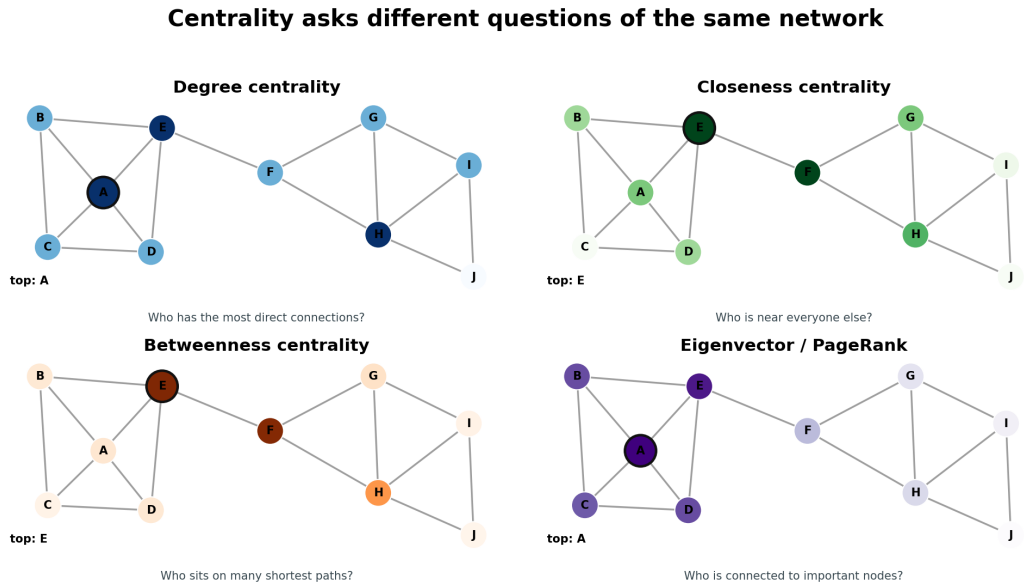


Figure 29.2: Different centrality measures ask different questions of the same network: direct connections, closeness to everyone else, shortest-path brokerage, or connection to already-important nodes.

Once the scores are in a data frame, they can be sorted for exploration or merged into a supervised learning table. For example, an analyst might test whether centrality predicts churn, fraud risk, recommendation uptake, or response to an outreach campaign.

Network resilience. Central nodes often matter because removing them changes the rest of the graph. A resilient network keeps most nodes reachable after a failure. A fragile network fragments when one high-value node disappears. We can quantify this by removing a node, then recomputing the largest connected component, average path length, or clustering coefficient.

```
hub = max(G, key=G.degree)
G_without_hub = G.copy()
G_without_hub.remove_node(hub)

largest = max(nx.connected_components(G_without_hub), key=len)
residual = G_without_hub.subgraph(largest)
print(len(largest), nx.average_shortest_path_length(residual))
```

Ego networks. An **ego network** focuses on one node and its immediate neighbourhood. A one-hop ego network keeps the chosen node, its neighbours, and the edges among those neighbours. This is useful when we want to understand a local context: whether a person is surrounded by one cohesive group, several disconnected groups, or a bridge between communities.

```
ego = max(G, key=G.degree)
```

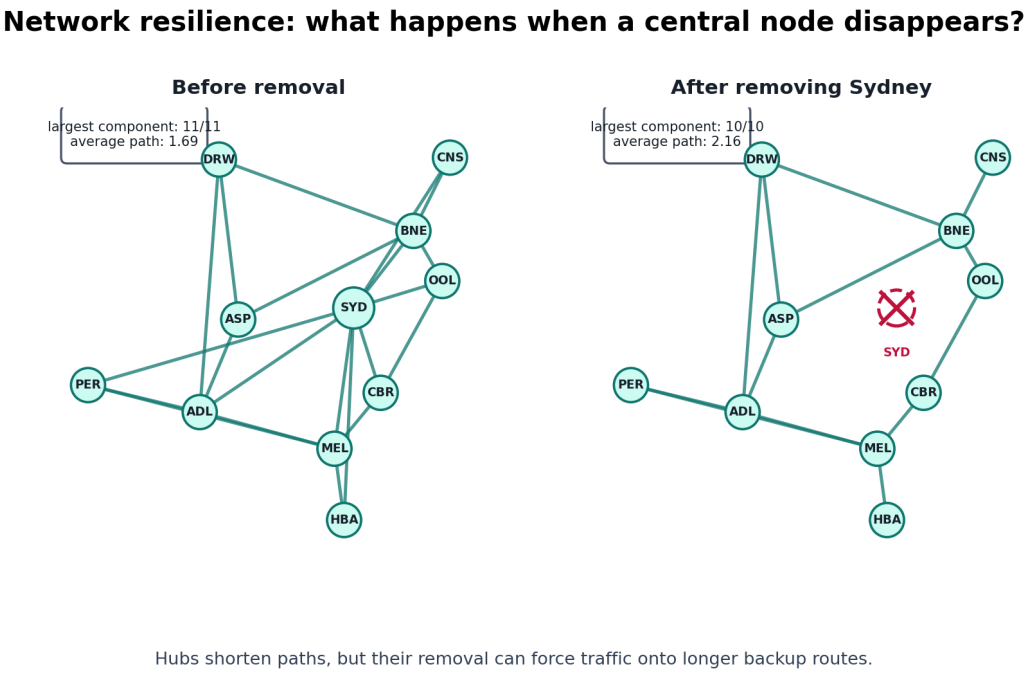


Figure 29.3: Network resilience asks what happens after an important node is removed. A targeted hub failure can fragment the graph and lengthen paths for the remaining nodes.

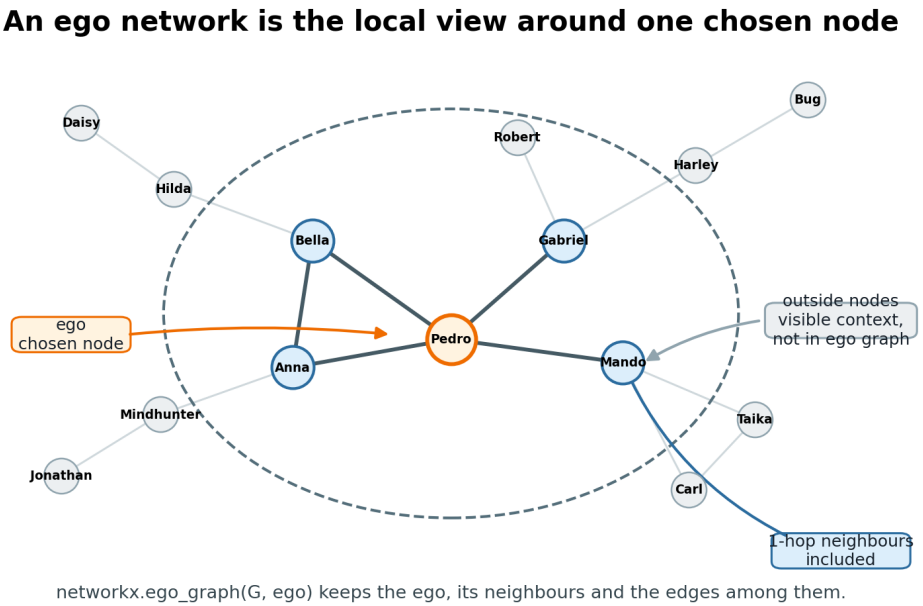


Figure 29.4: An ego network keeps one focal node and the surrounding neighbourhood. Comparing ego networks helps distinguish dense local groups from brokerage positions.

```
ego_net = nx.ego_graph(G, ego, radius=1)
nx.draw_networkx(ego_net)
```

Section summary

- Centrality scores encode different definitions of importance.
- Degree, closeness, betweenness, eigenvector/PageRank, and current-flow centrality answer different operational questions.
- Resilience analysis removes nodes and measures how much the remaining network changes.
- Ego networks summarise the local context around one focal node.

Self-check questions

1. Which centrality measure would you use to find a bottleneck in a transport network, and why?
   Hint: think about routes that pass through the same transfer point.

2. Why might a node with modest degree still be strategically important?
   Hint: consider connections to already-important neighbours or bridges between communities.

29.6 Measure clustering and network density

Quantify how tightly nodes group together and how redundant the connections are.

Learning objectives

After this section we will be able to:

- define the **clustering coefficient**: the fraction of a node's neighbours that are also neighbours of each other;
- compute average clustering with NetworkX to characterise network cohesion;
- interpret high clustering as redundancy (failure-tolerant infrastructure, tight-knit communities) and low clustering as sparsity (hub-and-spoke topologies);
- recognise when clustering metrics inform network design or social-dynamics research;
- explain how modularity-based community detection finds dense groups with sparse bridges.

Prerequisites

Make sure we can:

- construct and visualise NetworkX graphs;
- understand basic probability (fractions, proportions);
- appreciate that different network topologies serve different purposes (robustness versus efficiency).

The **clustering coefficient** measures how much a node's neighbours connect to each other. If node A links to nodes B and C, and B also links to C, then A's neighbourhood forms a tight triangle. High clustering indicates redundancy: removing one edge leaves alternative paths. Low clustering suggests a hub-and-spoke structure where the central node mediates all connections.

Formally, for a node with k neighbours, there are $\binom{k}{2} = k(k-1)/2$ possible edges among those neighbours. The clustering coefficient equals the fraction of those edges that actually exist. A node with three neighbours that all link to each other has clustering 1.0 (a complete triangle). A node whose neighbours never link to each other has clustering 0.0 (a star pattern).

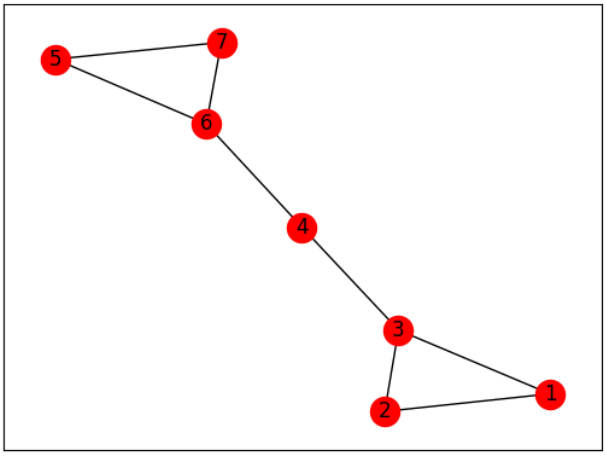


Figure 29.5: Local clustering asks whether a node's neighbours are also connected to each other. Triangles signal cohesive local structure; star-shaped neighbourhoods have low clustering.

Compute the average clustering coefficient across all nodes with:

```
avg_clustering = nx.average_clustering(G)
print(f"Average clustering: {avg_clustering:.3f}")
```

Values near 1.0 indicate a highly interconnected network where most nodes belong to tight cliques. Values near 0.0 suggest a sparse network dominated by hubs that connect disparate groups without internal cohesion.

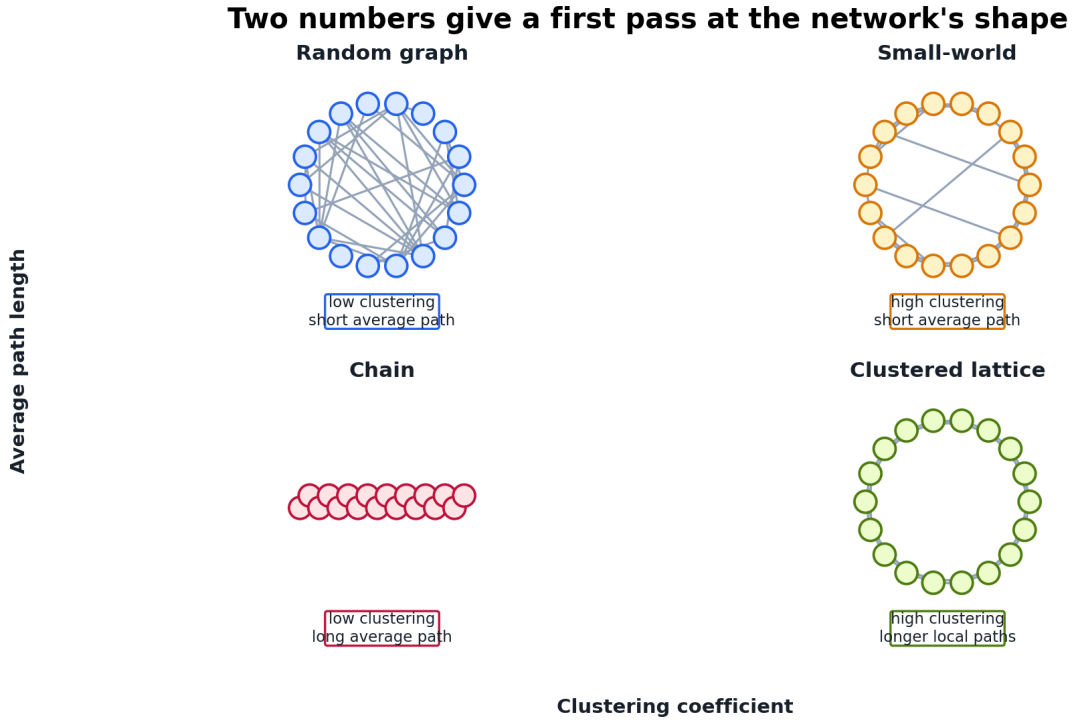


Figure 29.6: Networks with similar numbers of nodes can behave very differently. Average path length and clustering separate chains, lattices, random graphs, and small-world structures.

Applications.

- **Infrastructure resilience.** Electrical grids with high clustering tolerate line failures: alternative routes keep power flowing. Low clustering means single points of failure.
- **Social dynamics.** Friendship networks with high clustering exhibit strong community bonds; members' friends also know each other. Low clustering indicates bridging ties that connect otherwise separate groups.
- **Collaboration patterns.** Academic co-authorship networks with moderate clustering balance tight research teams (high local clustering) with interdisciplinary collaborations (low global clustering).

Lattice versus random networks. A **complete graph** connects every node to every other node. A **lattice** is different: it has an ordered local structure, such as a ring where each node connects to nearby neighbours. Lattices usually have high local clustering, but long average paths because travel proceeds through many local hops. A **random network**

(Erdős–Rényi model) connects nodes with fixed probability, producing lower clustering and shorter average paths.

Real-world networks often exhibit *small-world* properties: high clustering like lattices, but short path lengths like random graphs. A small number of long-range shortcuts can dramatically reduce average path length while preserving local neighbourhood structure. Scale-free networks are a different pattern: many nodes have low degree, while a few hubs have very high degree. The two patterns can coexist, but they answer different questions about how the network formed and how vulnerable it is to targeted hub removal.

Community detection. Once a network contains many dense local groups, we may want to find those groups directly. Community-detection methods look for regions with many internal edges and comparatively few bridges between groups. This is useful when we need to identify friend groups, collaboration clusters, loosely connected departments, or route islands that depend on a small number of bridges.

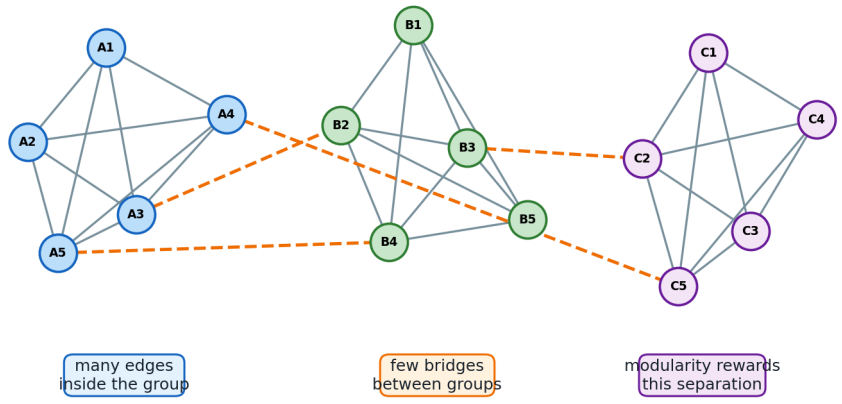


Figure 29.7: Community detection looks for groups with dense internal wiring and sparse bridges between groups. The dashed edges matter because they show how communities remain part of one wider network.

Louvain-style methods formalise this idea using **modularity**. Modularity asks how much more densely connected the proposed communities are than we would expect from a randomly wired graph with similar degree patterns. The algorithm starts with each node in its own community, moves nodes to neighbouring communities when that improves modularity, then collapses each community into a single super-node and repeats. The process stops when no move improves the score.

```
from networkx.algorithms import community
```

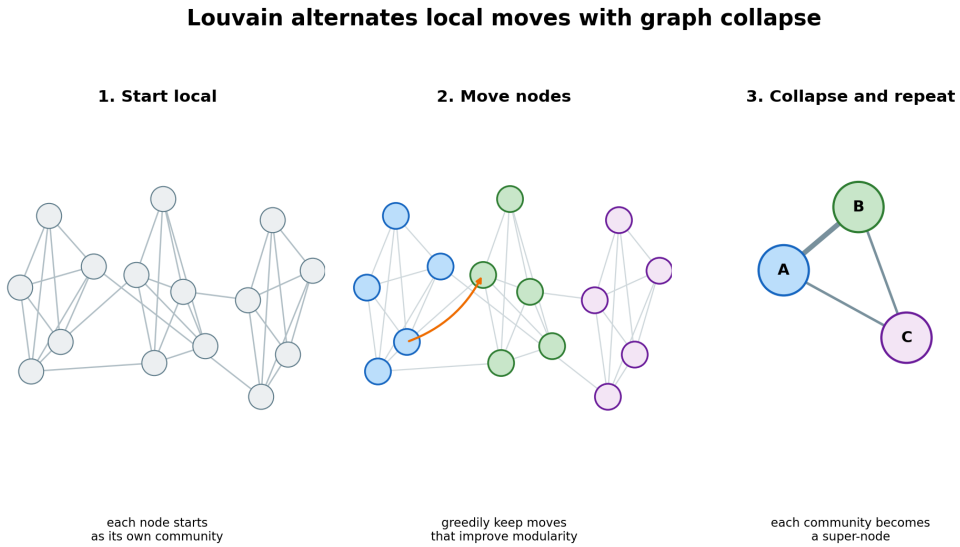


Figure 29.8: Louvain-style community detection repeatedly improves modularity, then collapses discovered communities and searches again at a coarser scale.

```
communities = community.louvain_communities(G, seed=42)
for group in communities:
    print(sorted(group))
```

For smaller or older environments, a greedy modularity function in the same NetworkX module gives a related result:

```
communities = community.greedy_modularity_communities(G)
```

Community labels can then be merged into the same node-feature table as centrality scores.

Section summary

- The clustering coefficient measures the fraction of a node's neighbours that link to each other.
- Compute average clustering with NetworkX.
- High clustering indicates redundancy and tight communities; low clustering suggests hub-dominated sparsity.
- Community detection finds dense groups linked by comparatively sparse bridges.
- Applications include infrastructure resilience, social-dynamics research, and collaboration-pattern analysis.

Self-check questions

1. How would you explain a clustering coefficient of 0.5 to a non-technical stakeholder managing a supply-chain network?
   Hint: frame it in terms of backup routes and alternative suppliers.

2. When designing a communication network, would you prioritise high clustering or low clustering, and why?
   Hint: consider trade-offs between redundancy (fault tolerance) and cost (number of links).

3. What does modularity try to capture when a community-detection algorithm groups nodes together?
   Hint: compare dense internal wiring with sparse bridges to other groups.

Concluding bridge

Network analysis extends data science beyond independent rows in a spreadsheet, modelling relationships that propagate influence, channel resources, and structure communities. This chapter introduced NetworkX for constructing graphs, spring layouts for visualisation, Dijkstra's algorithm for shortest paths, centrality measures for identifying important nodes, resilience checks for targeted removal, and clustering coefficients for quantifying network cohesion. The same toolkit applies to social-network datasets, where we might identify central influencers, compute average path lengths, and detect communities with modularity-based algorithms.

The next chapter (chapter 30) revisits collaborative filtering and implicit-feedback recommendation systems, where user–item interactions form bipartite networks. Techniques such as item-based similarity and matrix factorisation leverage network structure to predict preferences, demonstrating how graph thinking permeates modern personalisation algorithms. Together, network analysis and recommendation engines equip us to model interconnected systems where relationships matter as much as attributes.

Recognise that network metrics can serve as standalone insights ("Who are the top influencers?") or as engineered features ("Does centrality predict customer churn?"). Both use cases help answer diverse stakeholder questions, from operational decisions (optimise delivery routes) to strategic planning (redesign organisational communication flows). The same graph algorithms apply across social networks, supply chains, and biological pathways.

Chapter exercises

1. Decide whether several problems need network structure or ordinary tabular modelling. Justify one borderline case.

2. Build a small bipartite network from member names and group codes. Find one shortest path by hand, then recreate the same network in NetworkX.

3. Draw a spring layout twice with different seeds. Explain what the layout can and cannot prove.

4. Work through Dijkstra's algorithm by hand on a tiny weighted graph and record the final shortest path.

5. Compare degree, betweenness, clustering coefficient, and density for two small networks.

6. Pick one high-degree node, draw its ego network, and explain whether it looks like a tight local group or a bridge between groups.

7. Run a modularity-based community-detection algorithm and describe one bridge edge whose removal would separate two groups.

Chapter 30

Designing Recommendation Engines

Chapter overview

This is an optional computing-extension chapter. It begins with the practical question a recommender must answer: why this item, for this person, now? From there we compare simple baselines, content-based evidence, collaborative evidence, rules, context, and exploration. The second half keeps the advanced material: top-N ranking on implicit feedback, layered pipelines, sparse matrices, matrix factorisation, graph walks, and evaluation guard rails. The final section turns that material into a practical starting plan so we can ship a trustworthy baseline before exploring advanced models.

30.1 Start with the recommendation question

A recommender system chooses what to show next by combining evidence about items, users, past behaviour, and product goals.

Learning objectives

After this section we will be able to:

- define a recommender system in plain language;
- distinguish popularity, content-based, collaborative, rule-based, contextual, and exploration-based approaches;
- explain why the objective function matters before any modelling begins.

Prerequisites

We should be able to:

- read a small table of users, items, and ratings;
- interpret simple similarity ideas from chapter 11;

- discuss stakeholder goals and unintended consequences from chapter 1.

A recommender system is easy to recognise from the outside. It says, "we think you might like this." The useful engineering question is more precise: why this item, for this person, at this moment? A shopping site, a streaming service, a library catalogue, and a course-advice portal all answer that question differently because they optimise for different outcomes.

The simplest baseline is popularity. If we know nothing about a user, we can recommend the items with the highest average ratings or the most recent interactions. That can be a sensible fallback, especially at launch, but it is not very personal. A useful recommender should usually beat this baseline, or at least explain why it cannot.

Content-based recommenders use facts about items. If someone likes space adventures and fantasy adventures, we can recommend another title with matching tags, descriptions, actors, topics, or keywords. Content-based evidence is especially valuable for new items because it does not need other people to rate the item first. The risk is narrowness: if we only show more of the same, the system may never discover adjacent tastes.

Collaborative recommenders use behaviour patterns. They ask whether people who liked one set of items also liked another item, or whether items are connected through overlapping audiences. This is the intuition behind "people who liked this also liked that." Collaborative evidence can uncover surprising links, but it struggles when a new user or new item has no history.

Three extra building blocks keep the basic picture honest. Rule-based or constraint-based recommenders prevent impossible or unsafe suggestions: do not recommend a product that is out of stock, a medical action outside scope, or a degree plan that violates prerequisites. Context-aware recommenders recognise that the same person may want different items at different times, on different devices, or in different sessions. Exploration methods, including bandit-style policies, occasionally try uncertain options so the system can learn instead of repeating yesterday's safest answer.

Approach	**Evidence**	**Useful first question**
Popularity	Counts or average ratings	What should we show when we know almost nothing?
Content-based	Item features and metadata	What is similar to what this person already liked?
Collaborative	Shared behaviour across users and items	What did similar people or connected items reveal?
Rules and constraints	Policy, stock, eligibility, safety	What must never be shown?
Context-aware	Time, device, session, location	What does this situation change?
Exploration	Deliberate uncertainty	What should we try so the system keeps learning?

Context can be modelled as ordinary features, but a simple implementation can also

make context part of the profile. The same person on a phone at night may behave enough like a different user that the engine keeps a separate set of counts or similarities for that situation. This is especially useful when long-term history is unavailable because of privacy settings, account rules, or a new-device session. The recommender can still use the current item, the current session, transition patterns from other users, and popularity, but it should not treat an empty history as evidence of dislike.

Bandit-style exploration is the controlled version of trying an uncertain option. If the current recommendations are not working well, the system can allocate a small share of exposure to plausible alternatives and learn from the result. The important word is *controlled*: exploration needs limits, monitoring, and recovery rules so curiosity does not become random behaviour that frustrates users.

The product objective matters as much as the model family. A subscription service may prefer long-term satisfaction and retention. An advertising-funded feed may reward watch time, clicks, or session length. A public information system may value fairness, coverage, and trust over short-term engagement. The same data can therefore lead to different recommenders. If we optimise only for immediate attention, we can create filter bubbles, repetitive slates, or harmful feedback loops. If we optimise only for safety, the system may become stale and unhelpful. We need the objective and the safeguards in the project brief before we start tuning algorithms.

Section summary

- A recommender system chooses what to show next by combining evidence and product goals.
- Popularity, content-based, collaborative, rule-based, contextual, and exploration-based approaches solve different parts of the problem.
- The optimisation target shapes user experience, so safeguards belong in the design brief.

Self-check questions

1. A brand new catalogue has item descriptions but no user history. Which baseline and which personalisation method would you try first?
   Hint: separate launch-day coverage from later behaviour learning.
2. Why might an engagement-maximising video feed need diversity or safety rules?
   Hint: think about repeated exposure and feedback loops.
3. Which approach is easiest to explain to a user: popularity, content-based recommendation, or collaborative filtering? What does that explanation leave out?
   Hint: compare "many people liked it", "it shares these tags", and "similar users liked it."

30.2 Frame recommender problems around implicit feedback

Surface the right items by modelling top-N rankings, not just average ratings, while respecting catalogue constraints and cold-start realities.

Learning objectives

After this section we will be able to:

- distinguish rating prediction from top-N recommendation and justify why the latter dominates production settings;
- describe the structure of an implicit feedback log and the metadata that supports cold-start mitigation;
- outline the core product constraints — business rules, availability, and regulatory limits — that shape any recommender brief.

Prerequisites

We should be able to:

- read contingency tables and pivot interactions as reviewed in chapter 3;
- interpret model evaluation metrics from chapter 14;
- discuss how data governance policies influence modelling scope as introduced in chapter 1.

If you are new to the space, think of implicit feedback as the digital equivalent of noticing what someone lingers on in a shop. We rarely ask shoppers to score every product, yet we can still tell a story from their behaviour: they clicked on a raincoat, watched a how-to video all the way through, and added gardening gloves to the basket but never checked out. Those actions are soft signals rather than direct ratings, so we treat them as clues about interest rather than declarations of love or dislike.

In practice, explicit ratings remain rare. When customers purchase an item, they seldom leave a numerical score; if a venue asks patrons to write a review, perhaps one response arrives for every hundred visits. Instead, our logs contain binary observations: the customer chose chocolate yogurt but ignored the strawberry option. This binary choice reveals a preference ordering without exposing the underlying rating scale the customer might have imagined. We infer only that one item ranked higher than another, which suffices to power collaborative filtering and ranking models.

Recommender systems collect sequences of (u, i, t, c) tuples: a user u, an item i, a timestamp t, and optional context c such as device, locale, or intent. Most entries record implicit signals like clicks, plays, or basket additions instead of explicit star ratings. Because zero entries represent “unknown” rather than dislike, we reshape the problem into ranking

the most promising items for each user. Production teams align this to top-N lists: the ten videos to autoplay next, the five articles for a homepage shelf, or the bundle of elective subjects we want to surface in a portal.

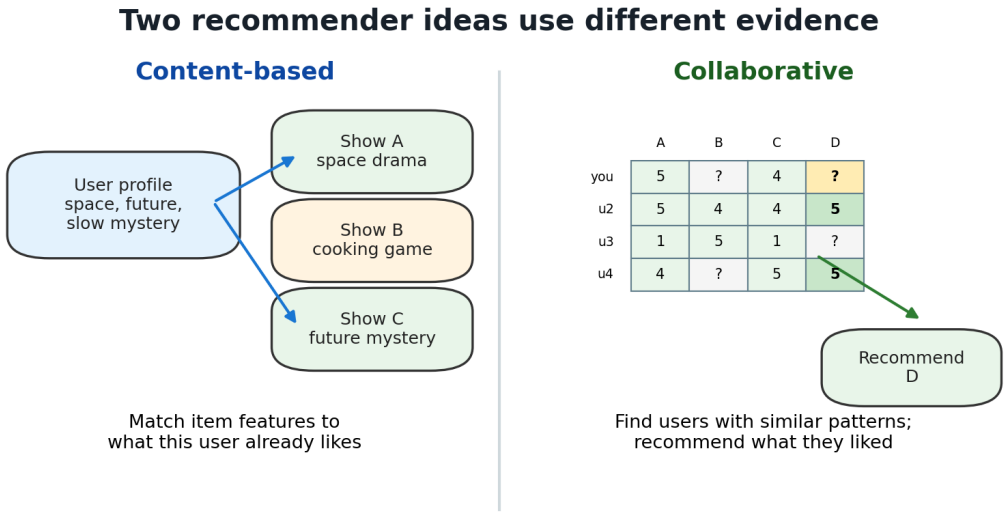


Figure 30.1: Content-based recommenders match item features to a user's known interests, while collaborative recommenders infer suggestions from similar behaviour across users and items.

Two structural challenges appear immediately. Cold-start users and items lack histories, so we borrow side information. Cohort details, declared interests, or onboarding questionnaire responses enrich user profiles, while item metadata — tags, textual blurbs, price points, and recency — helps the system reason about fresh catalogue entries. At the same time, long-tail behaviour means most catalogue items sit in sparse rows or columns. Rather than chasing full coverage with one algorithm, we layer signals: a global popularity fallback for coverage, similarity models to personalise mid-tail options, and content features for the newest additions.

Throughout, we respect non-technical rules. Merchandising teams may blacklist items awaiting clearance, policy teams prohibit certain combinations, and operations staff enforce stock limits. Document these constraints alongside the modelling objective before training. Doing so keeps later evaluation honest: a high hit rate on banned items is not a win.

Section summary

- Treat recommender tasks as ranking problems over implicit interactions rather than pure rating prediction.
- Enrich sparse interaction logs with user and item metadata to offset cold-start gaps.
- Encode business and compliance rules as first-class requirements before modelling.

Self-check questions

1. Which contextual attributes would you prioritise collecting to help a new streaming catalogue survive its launch month?
   Hint: balance demographic insight with privacy obligations.

2. How would you explain the difference between "no interaction" and "negative feedback" to a product manager planning KPIs?
   Hint: emphasise that implicit datasets rarely capture true dislikes.

3. A school careers portal wants to nudge students toward internships they might overlook. Which implicit signals could you log ethically, and how would you reassure stakeholders about data use?
   Hint: focus on opt-in browsing history, bookmarking, and transparent messaging.

Worked example: computing item similarity from implicit clicks

To demystify the mechanics, consider a toy implicit-feedback table with five learners browsing a skills platform. Each row records whether the learner viewed a resource in the last fortnight. The platform tracks ten resources: three videos (V1–V3), four practice sets (P1–P4), and three project briefs (B1–B3). We begin by pivoting the log into a binary user–item matrix X where $X_{ui} = 1$ if learner u interacted with item i.

Learner	V1	V2	V3	P1	P2	P3	P4	B1	B2	B3
L1	1	1	0	1	0	0	0	1	0	0
L2	0	1	1	0	1	0	0	0	1	0
L3	1	0	0	1	1	0	0	0	0	1
L4	0	1	1	0	0	1	0	0	1	0
L5	1	0	0	0	1	0	1	0	0	1

We now compute cosine similarity between two items — say V2 and B2 — by taking the dot product of their columns and dividing by the product of their norms. The column vectors are $v_{V2} = (1, 1, 0, 1, 0)$ and $v_{B2} = (0, 1, 0, 1, 0)$. Their dot product is $1{\cdot}0 + 1{\cdot}1 + 0{\cdot}0 + 1{\cdot}1 + 0{\cdot}0 = 2$. The vector v_{V2} has norm $\sqrt{1^2 + 1^2 + 0^2 + 1^2 + 0^2} = \sqrt{3}$, while v_{B2} has norm $\sqrt{0^2 + 1^2 + 0^2 + 1^2 + 0^2} = \sqrt{2}$, so the cosine similarity becomes $\frac{2}{\sqrt{3} \times \sqrt{2}} \approx 0.82$. A high score tells us that learners who watch V2 frequently explore B2 soon after.

Repeating this computation for every pair of items gives an item–item similarity matrix. When generating recommendations for learner L3, we look at the items they have already enjoyed (V1, P1, P2, B3) and pull the top neighbours from the matrix. Suppose B2 and P3 emerge with cosine similarities above 0.7 relative to the consumed items. We would recommend those next, while explaining to the stakeholder that the suggestions come from observing which resources tend to co-occur in similar study sessions. This exact workflow scales to thousands of items once we represent X as a sparse matrix, yet the core arithmetic mirrors this small example.

Worked example: counting paths in a user–item graph

The same idea can be drawn as a graph. Users and items become nodes, and a high rating or a like becomes an edge. A recommendation is a missing edge we might want to create. If Ben likes *Space Quest* and *Dragon School*, and Asha likes both of those plus *Robot Detective*, then there are two short paths from Ben to *Robot Detective*:

$$\text{Ben} \to \textit{Space Quest} \to \text{Asha} \to \textit{Robot Detective}$$

$$\text{Ben} \to \textit{Dragon School} \to \text{Asha} \to \textit{Robot Detective}.$$

Each path is weak evidence. Counting both paths gives *Robot Detective* a score of 2. If no similar path reaches *Baking Show*, then *Baking Show* scores 0 for this user under the graph method. This is collaborative filtering with counting rather than a complicated model.

Projecting the user–item graph into an item–item graph makes the same calculation easier to inspect. Two items are connected when at least one user liked both, and the edge weight records how many users provided that shared-neighbour evidence. We can then recommend items near the things the target user already likes. This links directly to the network ideas in chapter 29: degree, connected components, hubs, and path structure all affect the recommendation.

There is also a sampled version. Instead of counting every short path, we run many small random walks:

$$\text{user} \to \text{liked item} \to \text{another user} \to \text{another item}.$$

The endpoints that appear most often become recommendations. Random walks are noisy on a tiny dataset, but the idea scales naturally to large behaviour graphs where exact path counting is expensive. The practical risk is familiar from network analysis: popular hubs can dominate unless we add degree penalties, recency weights, diversity rules, or a second-stage ranker.

A collection-based platform gives a concrete version of the same walk. Start from an item a person saved, walk to the collection that contains it, walk to other items in that collection, and count the endpoints. This uses graph neighbourhoods rather than dense nearest-neighbour vectors, so it remains useful when most user–item cells are blank. It still needs popularity controls, because items that appear in many collections can otherwise swamp more specific recommendations.

30.3 Build layered recommendation pipelines

Start with interpretable baselines and gradually add collaborative, content-based, and ranking models to improve personalisation.

Learning objectives

After this section we will be able to:

- implement popularity and item–item co-occurrence baselines as dependable first passes;
- compare collaborative filtering, matrix factorisation, and content-based strategies for candidate generation;
- describe how two-stage architectures combine recall and ranking while enforcing business constraints.

Prerequisites

Before diving in, confirm that we can:

- compute cosine and Jaccard similarity scores as practised in chapter 3;
- manipulate sparse matrices or long-form interaction tables in pandas or NumPy;
- explain at a high level how gradient-based optimisation works from chapter 28.

Popularity and trending lists. Count interactions per item over a recent window to create a global popularity baseline. Segment the counts by cohort, geography, or device when different audiences behave differently. Apply a shrinkage factor (a weighted average that gently pulls noisy proportions toward a reliable global mean) $\hat{p}_i = \frac{n_i p_i + m\bar{p}}{n_i + m}$, where p_i is the item's observed proportion, so rarely observed items do not leap to the top due to a single lucky click. This baseline is not glamorous, yet it sets expectations, offers an immediate fallback when personalisation fails, and highlights coverage gaps.

Item–item co-occurrence. Convert the interaction log into a binary user–item matrix X where $X_{ui} = 1$ when user u touched item i. Compute similarities between item vectors with cosine or Jaccard metrics and score a target item i for user u by aggregating similarities to the items already consumed. Stakeholders appreciate the resulting "people who liked this also liked that" narratives, and the method thrives on implicit signals. Remind readers that very new items need help from metadata because no interactions exist yet.

Association rules for basket data. For checkout logs or short session histories, frequent-pattern mining provides high-confidence rules of the form $A \Rightarrow B$. Support, confidence, and lift reuse the same market-basket vocabulary that appears elsewhere in introductory analytics work. When we translate those metrics into recommendation scores, we keep the storytelling power of market-basket analysis while integrating it into the wider engine.

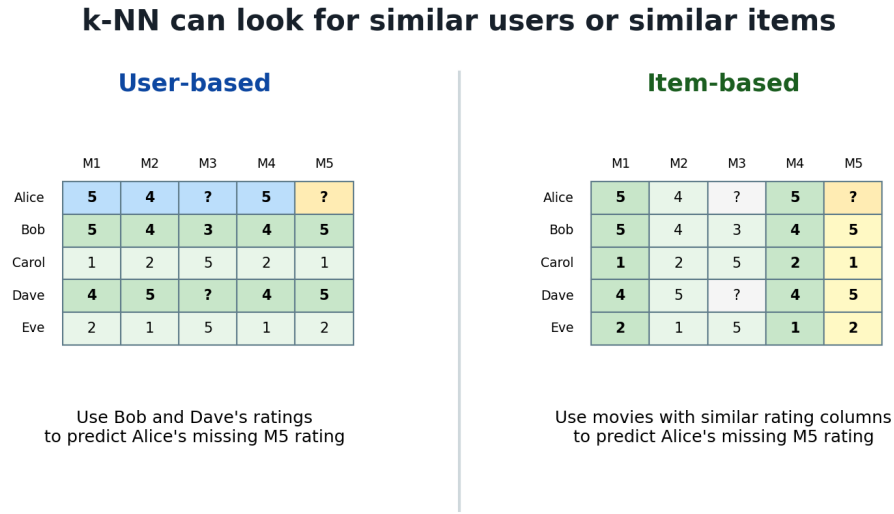


Figure 30.2: k-NN recommenders can compare similar users or similar items. Item-based neighbours are often more stable because catalogue relationships shift more slowly than individual user behaviour.

Two-stage architectures. Once baselines behave, combine them into a pipeline with distinct stages. Stage 1 recalls a tractable set of candidates using fast heuristics: popularity, co-occurrence neighbours, matrix factorisation factors, content-based similarity, or random walks on the user–item graph. Stage 2 ranks those candidates with richer features: personalised factors, recency scores, diversity penalties, or outputs from a gradient-boosted tree. A final post-processing pass enforces business rules, ensures a minimum share of long-tail items, and blocks banned content. Think of the pipeline as a conversation between shortlists and rerankers: the first ensures coverage, the second refines relevance.

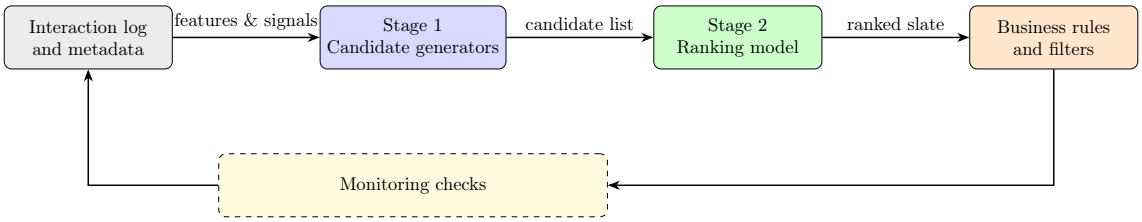


Figure 30.3: Two-stage recommendation flow.

Matrix factorisation for implicit data. Weighted regularised matrix factorisation (often delivered by alternating least squares) factorises the interaction matrix into user and item embeddings. We treat observed interactions as confidence-weighted positives and downweight the unobserved majority. The latent factors then serve two roles: they power a personalised dot-product score and supply features to downstream ranking models. Tune the regularisation strength (a penalty that discourages the factors from growing too large

and overfitting) and the number of latent factors so the model balances expressiveness with generalisation.

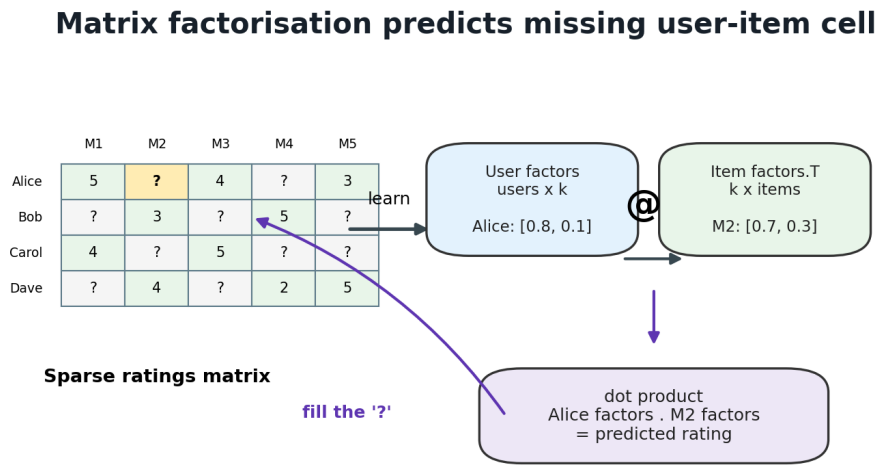


Figure 30.4: Matrix factorisation learns compact user and item factors, then uses their dot product to estimate missing user-item preferences.

Content-based and hybrid approaches. Vectorise item metadata — textual descriptions, tags, categories, or numeric attributes — and compute similarity directly in the feature space. This channel keeps recommendations flowing when catalogue additions arrive faster than user feedback accrues. Blend content and collaborative signals with weighted sums, stacking models, or learning-to-rank frameworks so the system honours both behavioural and descriptive evidence.

Pairwise ranking losses and graph walks. Techniques such as Bayesian Personalised Ranking optimise the relative order of preferred versus unobserved items, offering better top-of-list control than pure squared-error objectives. Personalised PageRank on a user–item graph, meanwhile, reuses network ideas from chapter 29 by running random walks with restart to prioritise items connected to the user's neighbourhood. These methods broaden the candidate toolkit once baselines and factor models are stable.

Section summary

- Ship popularity and co-occurrence baselines first to establish trust and provide fallback behaviour.
- Combine multiple candidate generators inside a two-stage architecture before introducing complex rankers.

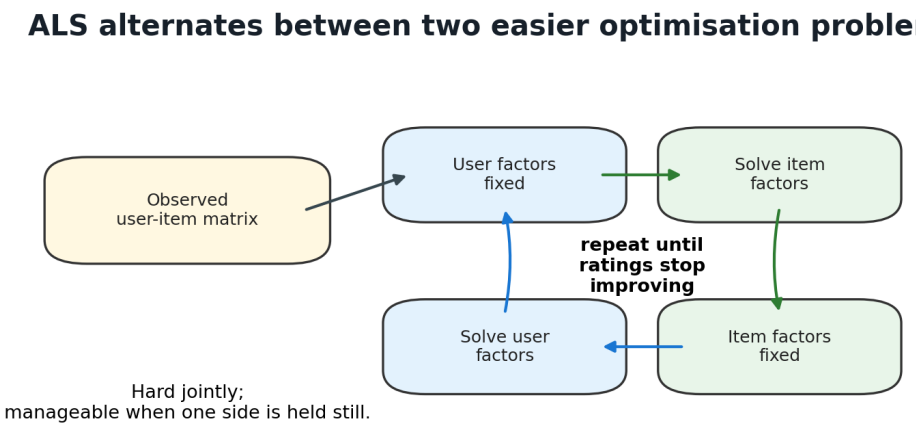


Figure 30.5: Alternating least squares makes matrix factorisation tractable by fixing one set of factors while solving for the other, then alternating until the ratings stop improving.

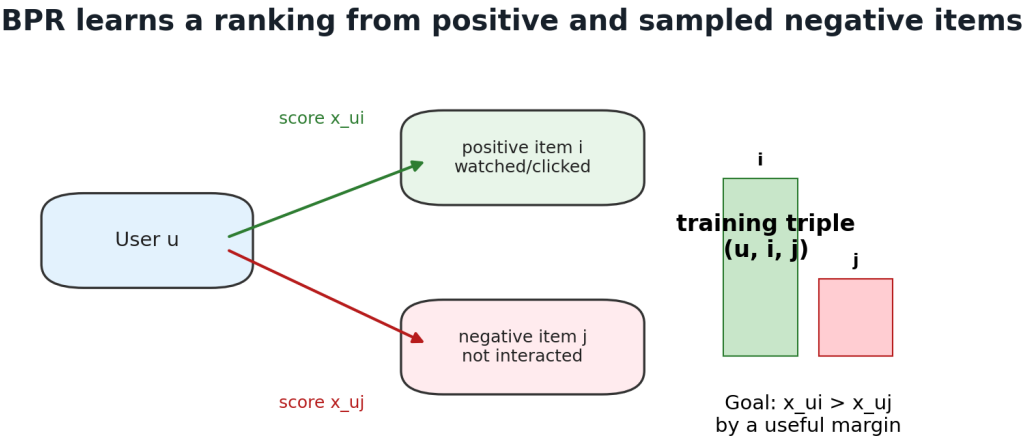


Figure 30.6: Bayesian personalised ranking trains on triples: a user, a positive item they interacted with, and a sampled negative item they did not. The goal is to score the positive item higher.

- Use matrix factors, content vectors, and pairwise objectives to personalise beyond simple heuristics while respecting operational rules.

Self-check questions

1. Your catalogue team launches 500 new items overnight. Which blend of candidate generators keeps them discoverable by the next morning?
   Hint: think about content vectors plus a popularity safety net.

2. How would you explain the benefit of a two-stage recommender to a stakeholder worried about compute cost?
   Hint: emphasise that the first stage narrows the search space before heavier ranking happens.

3. You notice that association-rule recommendations over-index on bundled accessories. How could you combine rules with similarity scores or diversification to balance the slate?
   Hint: revisit how post-processing and diversity penalties reshape final lists.

30.4 Evaluate, monitor, and govern recommendation engines

Match offline evaluation to real-world exposure, then track bias, coverage, and compliance risks once the model deploys.

Learning objectives

After this section we will be able to:

- construct temporal holdouts that respect the causal order of implicit interactions;
- select ranking metrics that align with product funnels and catalogue health;
- articulate bias and governance considerations, including feedback loops and do-not-recommend policies.

Prerequisites

We should already be comfortable with:

- creating train/validation/test splits without leakage as covered in chapter 14;
- interpreting ROC-style metrics and coverage statistics from chapter 18;
- collaborating with stakeholders to define acceptable trade-offs as practised in chapter 1.

Implicit logs capture behaviour over time, so random splits leak the future into the past. Instead, choose a cutoff timestamp T, train on interactions up to T, validate on the next slice, and reserve the most recent window for last-mile checks. This temporal discipline aligns the offline experiment with the deployment scenario: we always predict tomorrow using yesterday's knowledge. Track catalogue coverage alongside accuracy so we notice when only the headliners receive exposure.

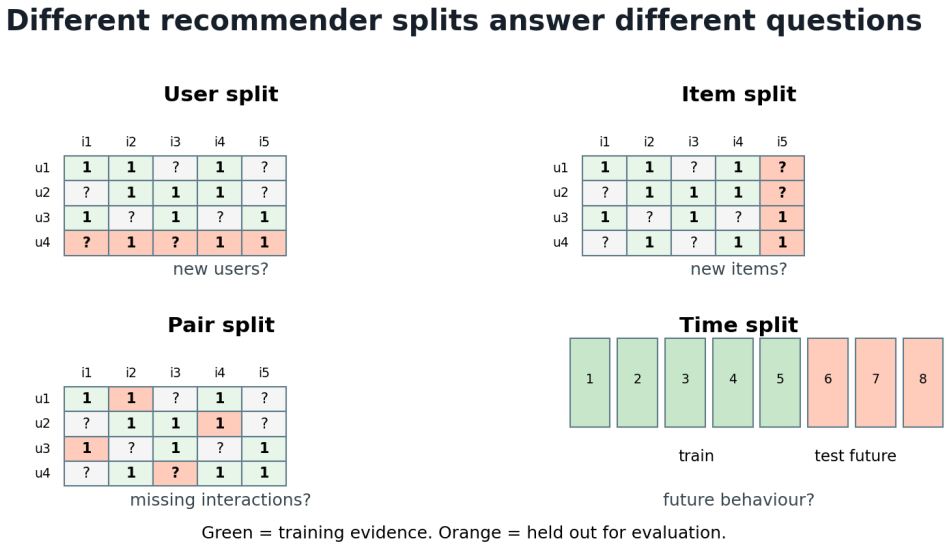


Figure 30.7: Different recommender splits answer different questions. A time split best matches production when the task is to predict future behaviour from past interactions.

Naïve random splits produce misleading validation scores for recommenders. Consider a user who has watched seasons one, two, three, five, and six of a series. If season four lands in the test set by chance, the model will trivially predict it belongs on the user's shortlist, inflating the metric without demonstrating genuine discovery or preference modelling. For catalogue-discovery questions, the holdout may also need to be grouped: remove the whole show, series, or product family, then ask whether the system can surface the first item from that group. Moreover, consumption behaviour shifts: after completing several seasons, viewers often seek thematically related content rather than the next episode of the same show. A temporal split captures that evolving preference; a random split conflates initial interest with sustained engagement.

Evaluation also has an intervention problem. A recommender changes the world it later measures. If the system recommends a scrubbing brush and the customer buys one, recommending the same item again may be pointless because the earlier recommendation changed the customer's need. This is a form of **algorithmic confounding**, also called **feedback bias**: the logs show behaviour under the old recommendation policy, not what users would have done without it. Good evaluation records exposure, tracks repeated-item usefulness, and reserves enough exploration traffic to estimate what the model is missing.

Choosing the right objective. Before selecting ranking metrics, clarify what the recommender should optimise. Streaming-subscription platforms and advertising-driven services pursue fundamentally different goals. A subscription service that charges a flat monthly fee benefits from member retention and satisfaction; whether a viewer watches ten minutes a day or five hours, the revenue stays constant. In contrast, platforms that monetise through advertisements earn more when users remain engaged longer, because every additional minute watched translates to more ad impressions. The latter optimisation can create filter bubbles that show only similar content, reducing serendipity, or drive rabbit-hole effects in which progressively more extreme recommendations maximise the probability of continued viewing.

Ethical recommender design therefore pairs the business metric with safeguards: inject diversity, limit consecutive recommendations in the same narrow topic cluster, and rotate long-tail content to prevent over-concentration on popular items. Stakeholders should recognise the scale involved. When a service reaches hundreds of millions or billions of users, the recommendation code shapes global media consumption patterns and carries responsibility for preventing harmful feedback loops.

Ranking metrics. Precision@k and Recall@k link naturally to top-N shelves; normalised discounted cumulative gain (NDCG) rewards correctly ordering relevant items. Beyond accuracy, monitor novelty via the mean negative log popularity and assess diversity by measuring the average dissimilarity between recommended items. When offline metrics improve, corroborate them with online experiments such as A/B tests or interleaving so the product team trusts the gains.

Bias and governance deserve equal space in the evaluation plan. Popularity bias can snowball: showing the same hits over and over suppresses the long tail. Feedback loops mean offline logs reflect past policies rather than counterfactual behaviour. Mitigation strategies include injecting exploration traffic, estimating counterfactual propensity weights, or rotating long-tail content into the shortlist. Keep a register of do-not-recommend rules covering safety, compliance, and stock availability, and ensure the post-processing stage enforces them. Finally, co-design explanation strings with stakeholders so end users understand why an item surfaced, which supports accountability conversations.

Section summary

- Temporal splits mirror deployment reality and prevent future information leaking into training.
- Combine precision/recall with coverage, novelty, and diversity metrics to balance accuracy with catalogue health.
- Monitor bias, feedback loops, and policy constraints as integral parts of the recommender lifecycle.

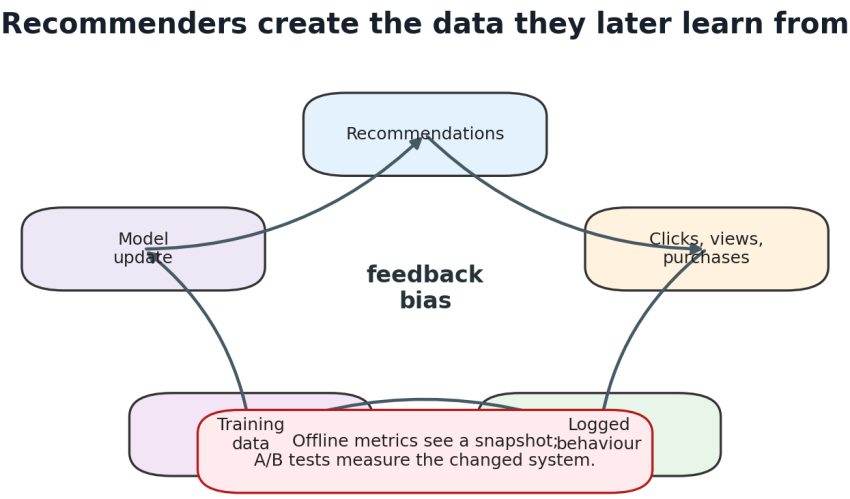


Figure 30.8: Recommenders create the data they later learn from. Live experiments and monitoring are needed because offline logs are snapshots of behaviour under an earlier recommendation policy.

Self-check questions

1. How would you convince a teammate to adopt a temporal validation split when a random split reports higher accuracy?
   Hint: highlight the risk of leaking future interactions and overestimating performance.

2. Which monitoring dashboard widgets would reveal a recommender drifting toward popularity bias over time?
   Hint: think about long-tail coverage and novelty trends.

3. Your compliance lead asks for proof that banned items never surface. Which offline tests and live alerts would you set up to provide that assurance?
   Hint: combine post-processing unit tests with production incident logging.

30.5 Turn the chapter into a practical starting plan

Arrive at the first build with a baseline, a shortlist of datasets, and an evaluation script so you can focus on iterative improvement.

There are three sensible entry points. Concept-focused readers can follow a notebook-based path: convert high ratings into like edges, build a user–item graph, project it into an item–item graph, and test the recommender by hiding one known like. Analytics-focused readers can pivot implicit logs into user–item matrices with pandas or spreadsheets, compute item–item similarities, and report Precision@10 alongside catalogue coverage. Programming-focused readers can implement sparse k-nearest neighbours, train an implicit

alternating-least-squares model, and compare NDCG@10 across a temporal split. All three tracks share the same rhythm: establish a popularity baseline, validate a personalised model, and document trade-offs in a project journal.

Prepare by selecting a dataset that fits your hardware budget. The small helper dataset in `simple_recommenders.py` is enough for explaining popularity, content matching, shared-neighbour paths, random walks, and hidden-item evaluation. MovieLens 100K offers quick iteration once the toy example feels clear; a small e-commerce clickstream sample highlights purchase funnels; open-education logs complement association-rule storytelling. Sketch a notebook or script template that loads the data, constructs the split, computes baseline metrics, and leaves placeholders for candidate generators you want to test. Bring a checklist of business rules that could apply in your chosen context so you can rehearse how to encode them in post-processing.

Next steps. Rehearse presenting a baseline dashboard to a stakeholder. Include the metric summary, a coverage snapshot, a short explanation of the recommendation evidence, and a paragraph on policy compliance. That preparation makes the first iteration smoother and mirrors real reporting expectations.

Chapter exercises

1. Convert a small user-item interaction log into a binary matrix. Explain why zeros mean "unknown" rather than "dislike."

2. Compare popularity, content-based, and collaborative-filtering baselines for a toy platform.

3. Draw a small user–item graph and count three-hop paths from one user to two candidate items.

4. Design a temporal train/test split that avoids using future interactions to predict earlier behaviour.

5. Hide one known like from a toy dataset and explain why recovering it is useful but not enough to prove the recommender is good.

6. Compute or interpret precision@k and recall@k for a short recommendation list.

7. Write a governance memo naming one fairness risk, one feedback-loop risk, and one monitoring metric.

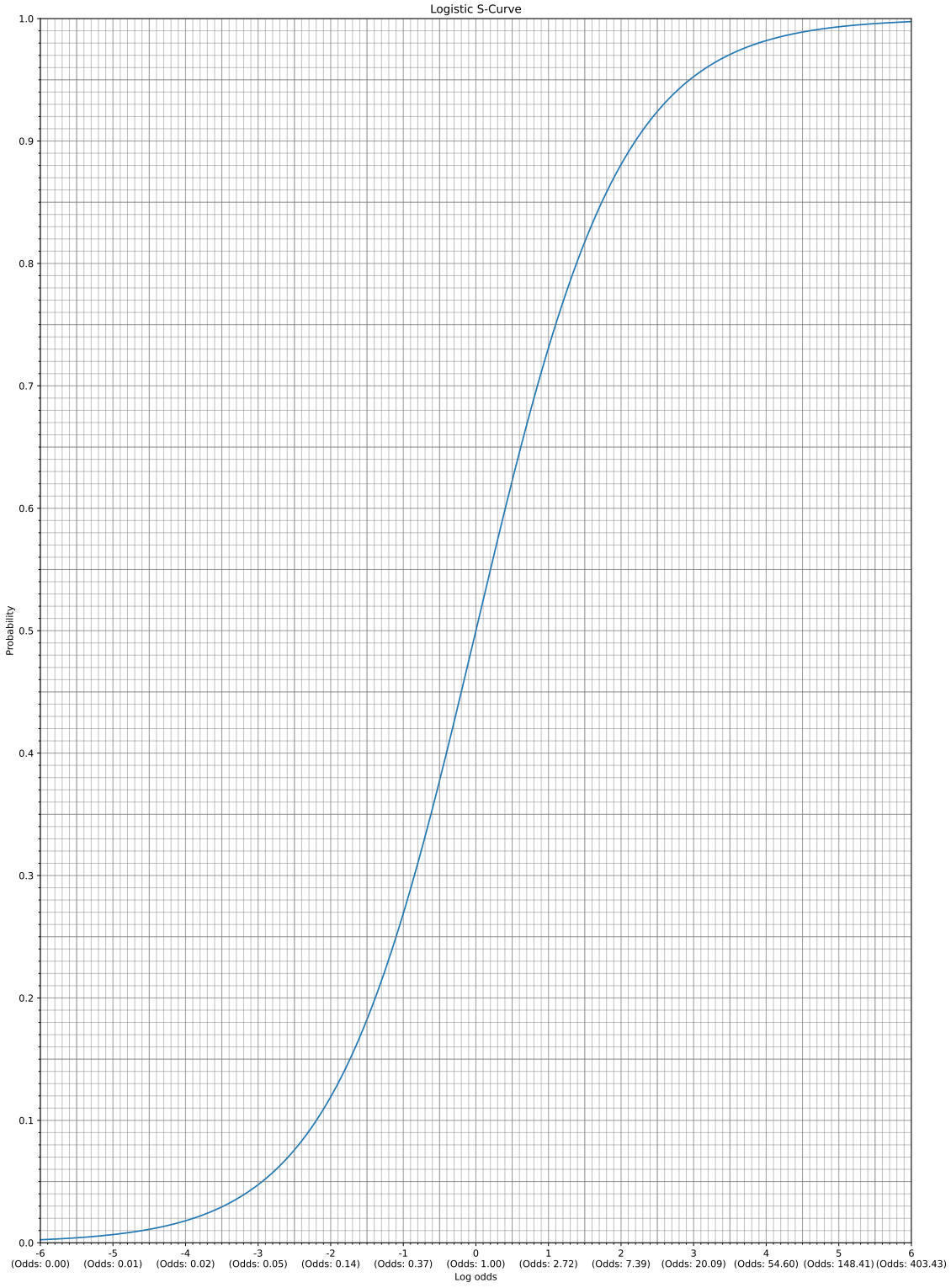
Appendix A

Practical Printables

This appendix collects the reusable print-ready artefacts that support the hands-on practical kits. If a tutor needs to reprint a persistent classroom aid, there should be a copy in the textbook as well as in the source folders.

A.1 Hands-on Activity 3: Logistic Curve Poster

The following page reproduces the logistic S-curve reference sheet used in the Week 3 chessboard activity. The source asset remains at `tools/logistic_s_curve.pdf`; including it here makes the poster discoverable from the textbook itself.


Logistic S-Curve
Probability
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
-6 (Odds: 0.00)
-5 (Odds: 0.01)
-4 (Odds: 0.02)
-3 (Odds: 0.05)
-2 (Odds: 0.14)
-1 (Odds: 0.37)
0 (Odds: 1.00)
1 (Odds: 2.72)
2 (Odds: 7.39)
3 (Odds: 20.09)
4 (Odds: 54.60)
5 (Odds: 148.41)
6 (Odds: 403.43)
Log odds

A.2 Hands-on Activity 5: Watchband Card Deck

The following pages reproduce the Week 6 watchband card deck in the A4 layout used for printing and guillotine cutting. The source asset remains at `EPIC/DecisionTrees/decision-trees-A4.pdf`; the alternate A5 layout still lives alongside it in the source tree.

Age: 14

Watchband colour: orange

Income: $10

Dollar Value

Age: 18

Watchband colour: orange

Income: $10 000

Dollar Value

Age: 12

Watchband colour: orange

Income: $0

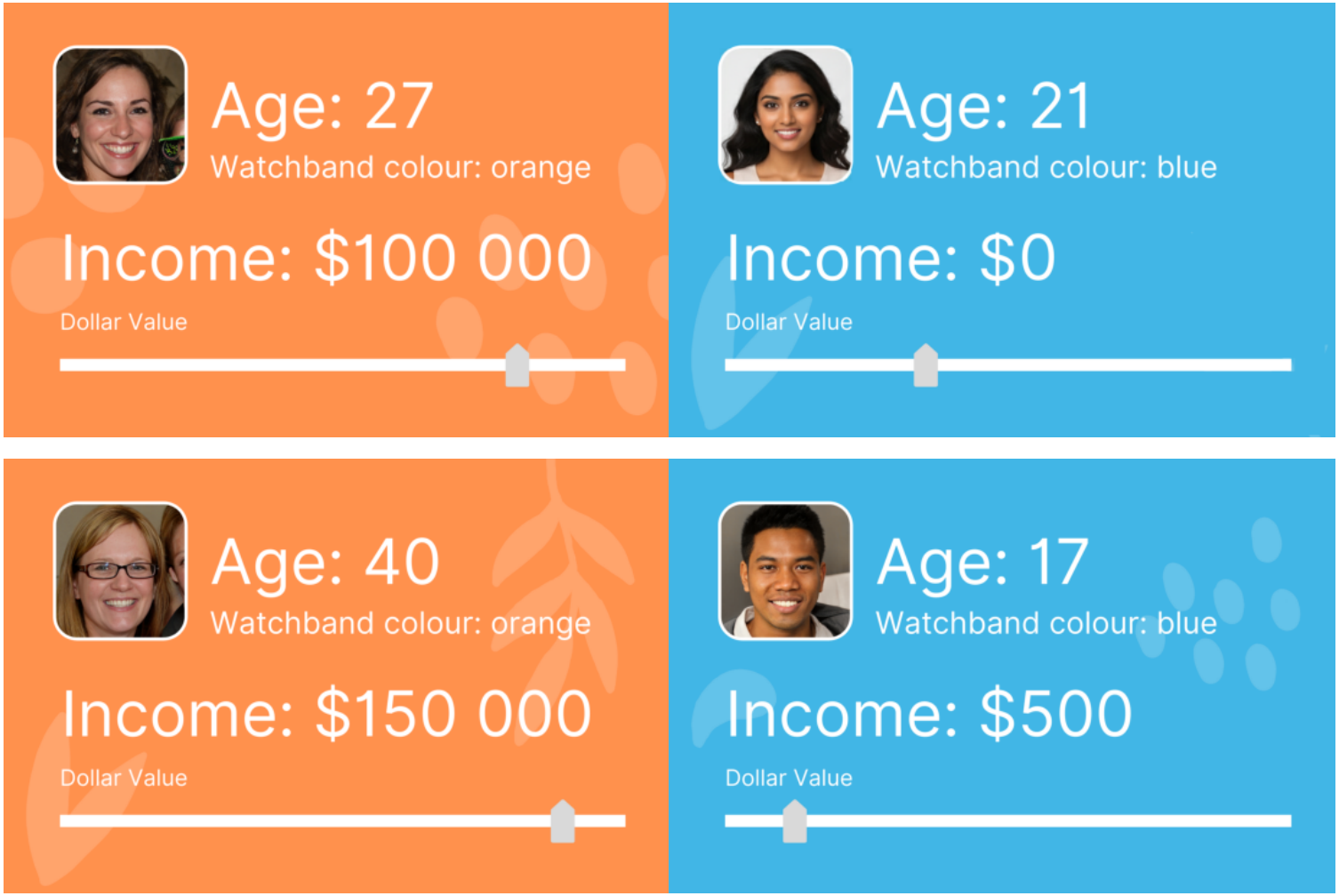
Dollar Value

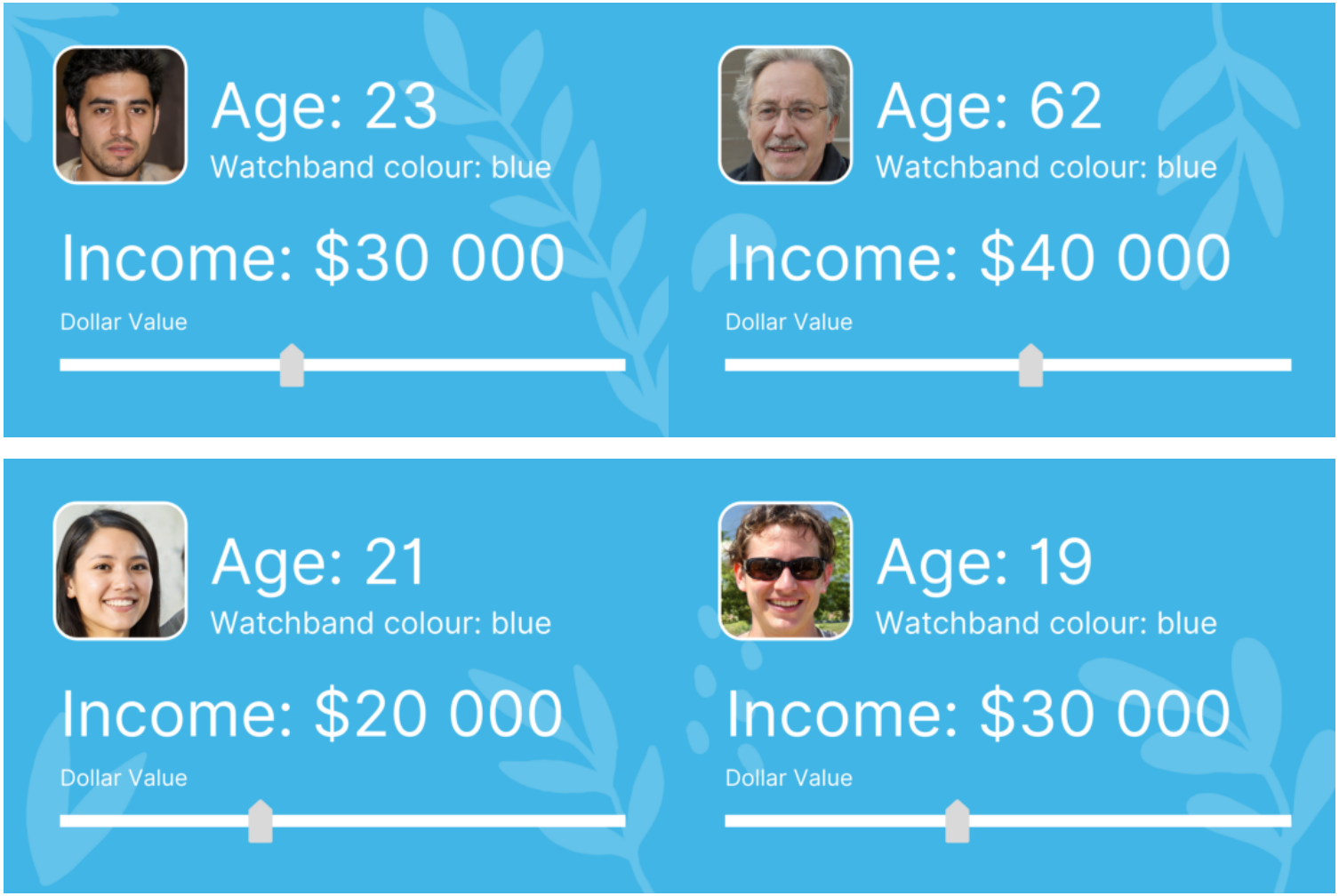
Age: 15

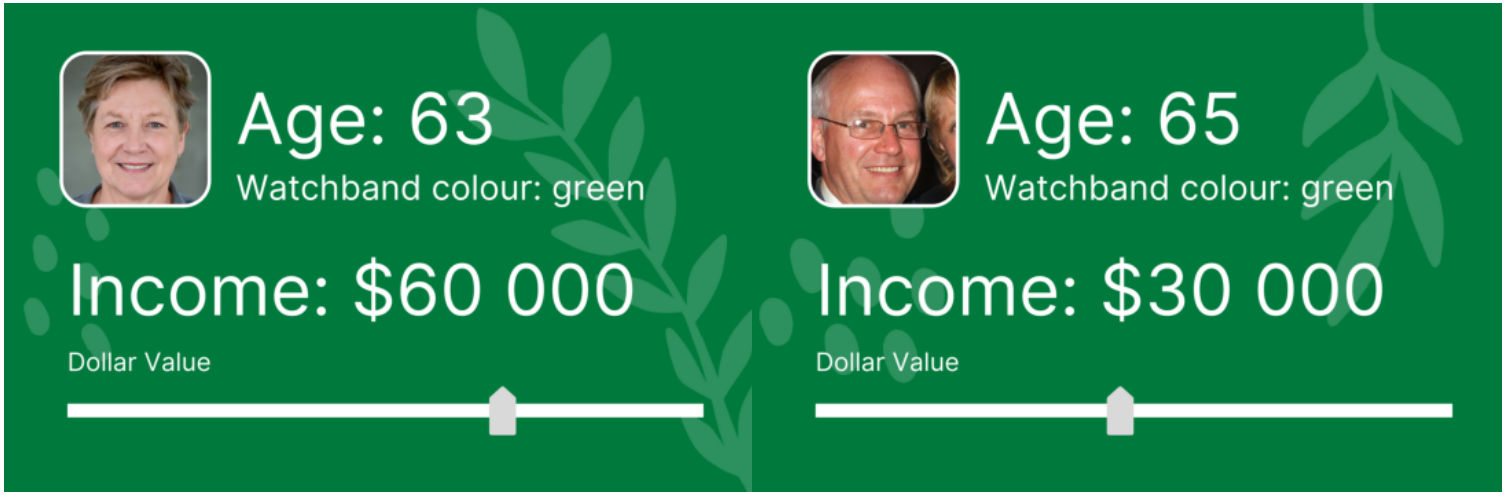
Watchband colour: orange

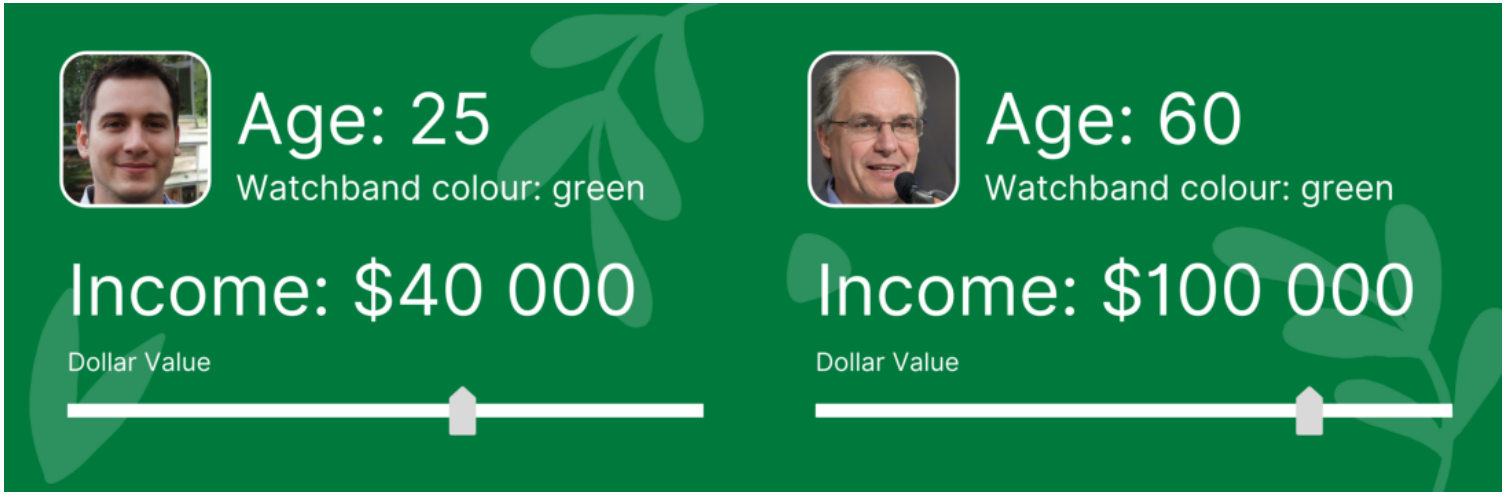
Income: $0

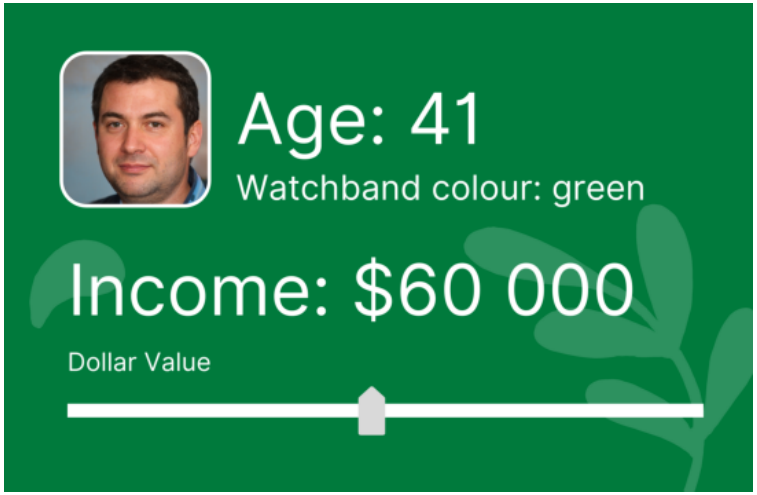
Dollar Value

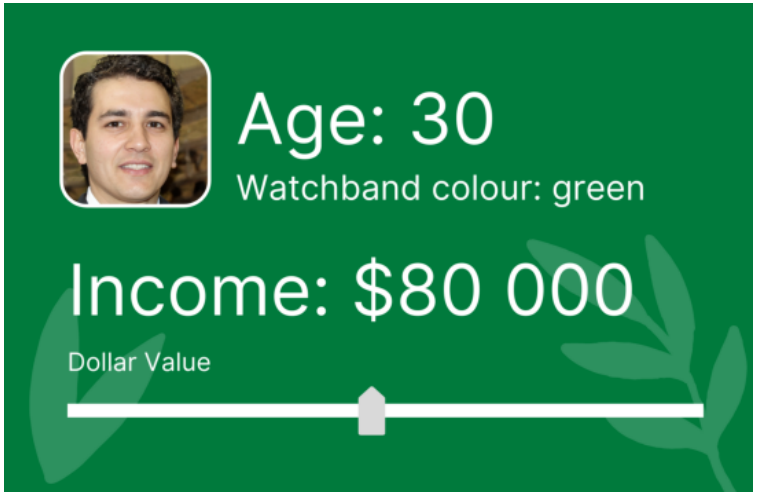

Age: 27
Watchband colour: orange
Income: $100 000
Dollar Value
Age: 21
Watchband colour: blue
Income: $0
Dollar Value
Age: 40
Watchband colour: orange
Income: $150 000
Dollar Value
Age: 17
Watchband colour: blue
Income: $500
Dollar Value


Age: 23
Watchband colour: blue
Income: $30 000
Dollar Value
Age: 62
Watchband colour: blue
Income: $40 000
Dollar Value
Age: 21
Watchband colour: blue
Income: $20 000
Dollar Value
Age: 19
Watchband colour: blue
Income: $30 000
Dollar Value


Age: 63
Watchband colour: green
Income: $60 000
Dollar Value
Age: 65
Watchband colour: green
Income: $30 000
Dollar Value


Age: 25
Watchband colour: green
Income: $40 000
Dollar Value
Age: 60
Watchband colour: green
Income: $100 000
Dollar Value


Age: 41
Watchband colour: green
Income: $60 000
Dollar Value


Age: 30
Watchband colour: green
Income: $80 000
Dollar Value

Solutions to Selected Exercises

These sketches give the expected shape of a good answer. Many textbook exercises ask for workflow evidence, judgement, or a short recommendation, so a strong answer may differ in details while still meeting the rubric.

Data Science Roles

1. A credible delivery-window project separates the work: analysts define late deliveries and inspect patterns, data scientists test predictors, data engineers stabilise source tables, machine-learning engineers monitor any deployed scorer, and domain experts explain dispatch constraints.

2. A good churn handoff preserves the original question, data definitions, split rule, model evidence, deployment owner, and monitoring trigger. Context is often lost between cleaning and modelling when excluded rows are not documented.

3. Strong rewrites name an outcome: which customers are likely to buy again, which product bundles increase basket value, or which regions respond to promotions. Each needs a concrete table, time window, and success measure.

4. Portfolio artefacts should be role-specific: a dashboard and memo for analysis, a model comparison notebook for data science, a documented data pipeline for engineering, or a monitoring plan for deployment work.

Tableau for Rapid Visual Exploration

1. Dimensions usually include identifiers, categories, regions, and dates; measures usually include price, count, distance, and revenue. Numeric codes such as postcode or product id often need manual checking.

2. The three worksheets should answer distinct questions: composition, trend, and relationship. A good answer names the field mappings rather than only saying "I made a chart."

3. The dashboard should lead with the chart closest to the stakeholder decision, then place supporting breakdowns below or beside it. Filters should be visible only when

they support the question.

4. PDF is usually better for vector print output; PNG is usually safer for slides and screenshots. Filenames should include dataset, chart, and date or version.

5. Tableau wins for fast interactive exploration; Python wins once the cleaning, model, or repeated export needs to be versioned and rerun.

Introduction to Orange

1. The `.ows` file answers "what workflow did we use?"; the CSV answers "what data moved forward?"; the saved model answers "what fitted object scored new rows?"; screenshots and reports preserve evidence for people.

2. The staging strip should load the file, assign target/feature/meta roles deliberately, and show the cleaned table. Notes should explain a real decision, not just restate the widget name.

3. Good observations include a pattern, a possible defect, and a next question. For example, high prices may cluster near the city, missing values may concentrate in one suburb, and time effects may need inspection.

4. Modelling begins once the canvas includes a learner, a split or validation step, metrics, and a reportable comparison.

Hands-on Activity 1

1. The cluster log should include the chosen k, a short reason, and a borderline point. The exact clusters can vary.

2. The Orange mirror should contain **Paint Data**, **k-Means**, **Scatter Plot**, and preferably **Silhouette Plot**.

3. Silhouette evidence supports a choice when most points have positive scores and few sit near zero or negative values.

4. Stretching one axis changes distance. Points separated mostly along the stretched axis become farther apart and are more likely to change assignment.

Retail Analytics Case Study

1. A good decision question names the action, such as where to place depots or territories. Risks include road access, lease constraints, incomplete addresses, and local demand.

2. Store latitude/longitude and demand proxies are features; names and addresses are usually metadata; identifiers are usually excluded from clustering.

3. Trustworthy checks include map inspection, duplicate coordinate checks, and suburb sanity checks. Local knowledge is still needed for road access and real travel time.

4. Alternative algorithms may change whether outliers are forced into territories. A useful answer says whether that affects the recommendation.

5. The memo should include a recommendation, evidence, uncertainty, and a validation step before capital is committed.

Exemplar-Based Clustering

1. Good similarity codes are repeatable and visible to another reviewer, such as incident type, customer intent, or product use case.

2. Exemplars should represent recognisable patterns, not merely extreme or amusing cases.

3. Early separation under repeated partitions suggests weak similarity; repeated co-location suggests stronger similarity.

4. Borderline records should be flagged rather than forced. The extra evidence should relate to the decision question.

5. Exemplar clustering helps when examples carry mixed evidence. It misleads when the exemplar pool is biased or when numeric distance already captures the structure cleanly.

Hands-on Activity 2

1. A plausible baseline follows the main cloud but should note whether outliers pull it away from ordinary points.

2. The Theil–Sen approximation should compute several pairwise slopes and choose a middle slope, not the most convenient line.

3. A RANSAC candidate survives when it has a strong inlier count under a stated tolerance.

4. The Orange line may differ from the hand line because the software optimises a formal loss across all entered points.

Linear Regression and Robust Alternatives

1. OLS is suspicious when residuals are dominated by a few cases, fan out with scale, or show distinct subgroup patterns.

2. Removing a high outlier usually pulls the line downward near that region; removing a leverage point can rotate the line.

3. Ordinary least squares suits clean linear data; Theil–Sen suits ordinary-case stability; RANSAC suits a clear main trend plus outliers; data cleaning comes first when records are wrong.

4. A strong report says what changed and why it matters for the decision, not that robust regression is universally better.

Hands-on Logistic Regression

1. The coordinate table should identify x and y as features and colour as the target.

2. Substitute coordinates into the linear score, then apply the logistic curve. Positive scores should produce probabilities above 0.5.

3. A coefficient update is useful only if the overall likelihood or error pattern improves, not just one favourite point.

4. If false positives are costly, choose a threshold that reduces false positives even if recall drops.

5. Extra features help only when the pattern is not linearly separable in the original coordinates.

Logistic Regression and Model Diagnostics

1. Warning signs include repeated tuning on the same validation output, adding weak features after seeing metrics, and opening the hold-out set early.

2. L1 often drives weak coefficients to zero; L2 shrinks them more gently. Shrinkage suggests the original model may have been chasing noise.

3. Compute metrics from the confusion matrix and choose the lead metric from the cost of errors.

4. Good calibration means cases scored near 0.8 are positive about 80 percent of the time.

5. A recommendation should mention ranking, threshold behaviour, probability quality, and a caveat about class balance or data drift.

Understand and Tune k-NN

1. Work distances explicitly. The 1-NN answer follows the closest point; the 3-NN answer follows the majority among the three closest points.
2. Rescaling changes which feature dominates distance. A changed neighbour ranking is expected when scales were uneven.
3. Choose k by balancing validation score, variability, and smoothness. The highest single score is not always the best choice.
4. Inverse weighting helps when closer neighbours are more trustworthy; it can amplify mislabeled or duplicated near-neighbours.
5. Staleness risks include changing geography, new products, privacy constraints, or query-time latency.

Predicting NSW Land Value

1. The valuation is the target; coordinates, area, and engineered location features may be predictors; parcel identifiers are metadata or excluded.
2. The query parcel must stay out of training data so the model does not evaluate itself on the row it is trying to predict.
3. A sensitivity table should show whether the predicted value is stable across k and weighting settings.
4. RMSE and MAE should be interpreted against the scale of land values and the decision being made.
5. Non-model checks include geocoding, zoning, recent policy changes, unusual land use, and comparable sales.

Red Rooster Line

1. Longitude scaling is a local approximation because degrees of longitude shrink with latitude; it is not a substitute for a projected coordinate system.
2. The shared evaluation node should receive all competing learners so the comparison uses the same folds or samples.
3. A strong answer mentions accuracy plus at least one of F1, ROC AUC, MCC, class balance, or confusion-matrix asymmetry.
4. Logistic regression assumes a smooth linear boundary; k-NN allows local irregularity but may overreact to isolated points.

5. The recommendation should cite metrics, geography, evidence files, limitations, and a refresh or monitoring plan.

Model Evaluation

1. Leakage includes fitting preprocessing on all rows, tuning after hold-out inspection, and letting target-derived fields enter the features.

2. A sound validation plan fixes folds, seeds, metrics, and hold-out rules before comparing models.

3. Baselines test whether complexity adds value. Constant, stratified, simple-rule, or popularity baselines suit different tasks.

4. Accuracy fails when classes are imbalanced or costs differ. Regression, ranking, and threshold tasks need different metrics.

5. A good lab log is reproducible: data version, split rule, models, parameters, metrics, chosen model, and caveat.

Hands-on Activity 4

1. The prediction log should separate true positives, false positives, true negatives, and false negatives or their activity equivalent.

2. Nearest-neighbour reasoning follows local examples; logistic reasoning follows the straight boundary and probability distance from it.

3. The Orange evidence should include the painted data, learners, **Test & Score**, and a saved metric table or screenshot.

4. `random_state` fixes repeatability; `test_size` controls the hold-out fraction. Test rows stay hidden to protect evaluation.

5. A complete evidence pack links the physical layout, workflow, metrics, and conclusion.

Explainable Models

1. Directly explainable features are familiar, stable, and meaningful. Engineered or proxy features need documentation.

2. Coverage says how many cases the rule touches; class distribution says what it predicts; WRAcc rewards useful precision without ignoring coverage.

3. A tree-path explanation should list the split conditions and the final prediction in domain language.

4. Choose transparency when accountability, policy review, or expert challenge matters more than a marginal metric gain.

5. Expert review should invite correction, identify the evidence needed, and avoid presenting induced rules as final truth.

Narrative Learning

1. A usable rulebook is short, deterministic, and testable by another group.

2. Revisions should point to validation mistakes, not general preference.

3. Test errors are recorded after the rulebook is frozen. Proposed improvements belong in a separate next-version note.

4. Audit findings often involve vague language, hidden priority rules, or exceptions that mirror the training examples too closely.

5. Governance evidence includes rulebook versions, training packets, validation mistakes, blind-test results, and reviewer comments.

Metrics and Ensembles

1. ROC construction follows the sorted score list and updates true-positive and false-positive rates as the threshold moves.

2. Threshold choice should minimise or justify expected cost, and the cost assumptions should be visible.

3. Useful diversity means the learner makes different errors for understandable reasons.

4. Prefer the simplest tuning setting that delivers stable improvement rather than the most complex high-water mark.

5. AUC ranks cases; deployment thresholds decide action. Both are needed.

Hands-on Activity 5

1. A split log should show class counts on each side and name the split that most reduces confusion.

2. Coverage is the number or proportion of records a rule applies to. WRAcc rewards rules that improve on the base class rate while covering enough cases.

3. Rule lists can be read clause by clause; trees show a single path clearly. The easier explanation depends on the stakeholder task.

4. The Orange comparison should use the same data and evaluation settings for tree, CN2, and random forest.

5. A balanced debrief can recommend a transparent model for explanation, a random forest for accuracy, or a two-stage compromise.

Python When It Helps

1. Strong choices mention constraints: Tableau for immediate stakeholder exploration, Orange for visible modelling, Python for reproducibility or unavailable methods.

2. Data files preserve rows and columns; workflows preserve GUI steps; notebooks/scripts preserve code; model files preserve fitted state.

3. A high-level Python translation should include load, inspect, select columns, transform, split, model, evaluate, and export.

Robust Regression in Python

1. Fix random seeds where possible and compare MAE/RMSE under the same folds.

2. Changing the RANSAC threshold changes which points count as inliers; too narrow can discard ordinary noise, too wide can include outliers.

3. The best resistant estimator is the one that improves the decision-relevant error, not merely the one with the neatest line.

4. The report should name the model, evidence, data issue, and whether the robust method changes the recommendation.

Probability Foundations

1. Frequentist statements describe long-run frequency; Bayesian statements update belief with evidence; informal statements often mix probability with confidence.

2. The random variable is the quantity that could vary, the sample is what was observed, and the parameter is the unknown population feature.

3. Larger samples produce stabler averages and smoother histograms, though individual draws remain random.

4. Sample means may look normal even when raw observations do not. Heavy tails, dependence, and tiny samples weaken the approximation.

5. A long right tail may be natural for multiplicative processes, but suspicious records and measurement errors still need checking.

NSW Road-Crash Data with pandas

1. The answer should record import choices, skipped rows if any, final shape, and obvious anomalies.

2. Single-column bracket selection often returns a Series; double brackets preserve a DataFrame.

3. Derived ratios must guard against zero or missing denominators and document excluded rows.

4. Pairplot observations should describe association and outliers without claiming causality.

5. The export answer should include a reproducible filename, shape, and hash.

Matplotlib Foundations

1. The expected pattern is `fig, ax = plt.subplots()`, followed by axis methods for plotting, labels, title, and legend.

2. The caption should explain encodings such as position, colour, and marker size.

3. Print needs sufficient physical size and resolution; slides often need a clear PNG at screen dimensions.

4. Pandas shortcuts are fast for exploration but can hide figure and axes control.

5. Tableau is useful for interactive stakeholder filtering; Matplotlib is stronger for scripted report generation.

Matplotlib Styling

1. Redundant encodings include marker shape, line style, labels, or direct annotation.

2. A fixed or honest baseline prevents exaggerated differences in bar charts.

3. Trend-line captions should mention extrapolation risk and that association is not causation.

4. Alt text should state the threshold values and labelled regions, not only the colours.

5. A figure review should check axes, labels, colour accessibility, baselines, caption, and export quality.

Pipelines in scikit-learn

1. A sound column audit separates predictors, target, metadata, and dropped fields before code is written.
2. Imputation choices should follow distribution and meaning. Missing is not always zero.
3. The pipeline should contain feature engineering, preprocessing, and estimator so validation repeats the same operations inside each fold.
4. Unsafe preprocessing is fitted before the split or on all folds. Move it inside the pipeline.
5. `model__C` names the `C` parameter inside the pipeline step called `model`; the double underscore follows the path through named steps.
6. The honest tic–tac–toe model uses only the moves available at prediction time. Including later moves leaks future information and answers a different question.
7. Reuse requires the fitted pipeline, feature schema, importable helper functions such as `engineer`, versions, training date, and expected input format.

Text as Data

1. Bag-of-words suits fast, transparent, auditable classification; LLMs suit nuanced language tasks where context matters.
2. Token decisions should separate obvious noise from domain terms that look odd but carry meaning.
3. Bigrams capture short phrases such as “free entry” or “call now” that unigrams may weaken.
4. Sparse matrices have thousands of mostly empty columns, so direct visual inspection is hard.
5. UMAP suggests neighbourhoods and outliers; it does not prove exact distances or causal clusters.

Text Model Tuning

1. An express grid should keep the number of combinations small enough for the environment and teaching time.
2. Choose from result tables by considering score, standard deviation, runtime, and simplicity.

3. Lowercasing merges variants; stop words remove common terms; n-grams add context but increase feature count.

4. A useful run log lets a teammate reproduce or deliberately skip the same experiment.

5. Add search dimensions that address observed error; avoid dimensions that only expand compute without a hypothesis.

Network Analysis

1. Use networks when relationships between entities affect the answer. Use tables when rows are mostly independent.

2. In the bipartite network, member slips and group-code slips are nodes, and the thread pieces are edges. A shortest path between two members should alternate through group nodes, and the NetworkX graph should have the same edges as the physical or spreadsheet version after blank group cells are skipped.

3. Spring layouts are visual aids, not geographic or causal proof. Different seeds may move nodes without changing graph structure.

4. Dijkstra's algorithm repeatedly fixes the closest unsettled node and updates neighbour distances.

5. Degree, betweenness, clustering, and density answer different questions about importance and cohesion.

6. An ego network is one node plus its immediate neighbours and the edges among them. A tight local group shows many edges between those neighbours (high clustering); a bridge shows few, with the central node connecting otherwise separate clusters.

7. Modularity-based detection groups nodes that are more densely connected to each other than to the rest of the graph. A bridge edge is one whose removal raises the number of components; cutting it separates two communities.

Recommendation Engines

1. Binary matrices record observed interactions. A zero usually means no observed interaction, not negative preference.

2. Popularity is a baseline, content uses item/user attributes, and collaborative filtering uses interaction patterns.

3. A three-hop path runs user → item → user → item. Count the distinct such paths from the starting user to each candidate item; the candidate reached by more shared-interest paths is the stronger recommendation, matching the path-counting example in the chapter.

4. Temporal splits train on earlier interactions and test on later ones so the evaluation resembles deployment.

5. Hiding a known like and checking whether the model ranks it highly is a quick leave-one-out sanity check. Recovering it is necessary but not sufficient: it ignores ranking of the full list, coverage, novelty, and cold-start behaviour, so pair it with the metrics in the next exercise.

6. Precision@k asks how many recommended items were relevant; recall@k asks how many relevant items were recovered.

7. Governance should mention exposure fairness, popularity feedback loops, banned items, drift, and catalogue coverage.

References

Software and documentation

[1] Orange Data Mining. *Orange Visual Programming Documentation*. `https://orange3.readthedocs.io/`.

[2] scikit-learn developers. *scikit-learn User Guide and API Reference*. `https://scikit-learn.org/stable/`.

Datasets and public data sources

[3] Horst, A. M., Hill, A. P., and Gorman, K. B. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. `https://allisonhorst.github.io/palmerpenguins/`.

[4] Harper, F. M., and Konstan, J. A. (2015). The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4), Article 19.

[5] Property NSW and the Valuer General of New South Wales. *Bulk Land Value Information Available on the Valuation Services Portal*. Data.NSW. `https://data.nsw.gov.au/data/dataset/327f9982-e610-4ffe-a51c-4047e97c7e7d`.

[6] Property NSW. *NSW Land Value and Property Sales Web Map*. Data.NSW. `https://data.nsw.gov.au/data/dataset/5df4d78a-e9b0-4216-b46d-8d87997e6cb4`.

[7] Transport for NSW. *NSW Crash Data*. Data.NSW. `https://data.nsw.gov.au/data/dataset/2-nsw-crash-data`.

[8] Transport for NSW. *Road Safety Statistics and Interactive Crash Statistics*. `https://www.transport.nsw.gov.au/roadsafety/statistics`.

Further reading

[9] Arthur, M. B., Inkson, K., and Pringle, J. K. (1999). *The New Careers: Individual Action and Economic Change*. SAGE.

[10] Lazear, E. P., and Rosen, S. (1981). Rank-order tournaments as optimum labor contracts. *Journal of Political Economy*, 89(5), 841–864.

[11] Åström, K. J., and Murray, R. M. (2010). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press.

[12] Perrow, C. (1999). *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press.

[13] Baker, G. D. (2025). *It's 2025 – Narrative Learning is the new baseline to beat for explainable machine learning*. Draft manuscript. `https://github.com/solresol/narrative-learning`.

[14] Ting, K. M., Wells, J. R., and Washio, T. (2019). *Isolation Kernel: The X Factor in Efficient and Effective Large Scale Online Kernel Learning*. arXiv:1907.01104. `https://arxiv.org/abs/1907.01104`.

[15] Ting, K. M., Wells, J. R., and Zhu, Y. (2020). *Point-Set Kernel Clustering*. arXiv:2002.05815. `https://arxiv.org/abs/2002.05815`.

[16] Han, X., Zhu, Y., Ting, K. M., and Li, G. (2023). The impact of isolation kernel on agglomerative hierarchical clustering algorithms. *Pattern Recognition*, 139, 109517. `https://doi.org/10.1016/j.patcog.2023.109517`.

Index