Introduction to Machine Learning

for Managers

George Georgiadis

Preface

These lecture notes were written for **AI Foundations for Managers – Strategy (AIML-901ST)**, an introductory machine learning course for MBA students at the Kellogg School of Management, Northwestern University. The course is designed for future general managers, strategists, and entrepreneurs—people who will commission, evaluate, and govern machine learning projects rather than build them from scratch.

These notes cover the four main branches of machine learning.

Part I introduces supervised learning: the problem of learning to make predictions from labeled examples. It begins with the end-to-end workflow—data preprocessing, model training, and evaluation—and then works through the major model families: linear and logistic regression, k-nearest neighbors, decision trees, ensemble methods (bagging and boosting), support vector machines, and neural networks, including specialized architectures for sequences and images. The part closes with model interpretability and practical guidelines for model selection.

Part II covers unsupervised learning: discovering structure in data without labeled outcomes. It treats clustering (K-means, hierarchical, and density-based methods), dimensionality reduction (PCA, UMAP, t-SNE), and autoencoders as tools for representation learning and anomaly detection.

Part III turns to reinforcement learning: the problem of learning *what to do* when actions today reshape the situations you face tomorrow. It develops the core concepts—states, actions, rewards, value functions, and the Bellman principle—through business examples, illustrates Q-learning with a worked Blackjack example, and surveys how these ideas scale to real-world applications in pricing, advertising, robotics, and operations.

Part IV covers natural language processing with large language models: the problem of extracting insight from unstructured text at scale. It opens with LLM foundations—tokenization, attention, embeddings, and prompt engineering—and then works through the major application areas: text classification, information extraction, semantic search, retrieval-augmented generation (RAG), and text clustering. The part closes with compliance and risk detection, which ties the preceding capabilities together in a high-stakes, human-in-the-loop setting.

The notes aim to strike a particular balance. They go deeper than a typical executive overview: formulas, algorithms, and loss functions appear throughout, because understanding *how* models learn is essential to knowing when to trust them and when to be skeptical. At the same time, they stay closer to intuition and business judgment than a computer science textbook would. Mathematical notation is introduced only when it clarifies the underlying logic, and methods are discussed not just in terms of how they work, but in terms of when they are useful, what can go wrong, and what a decision-maker should watch for. The goal is not to train data scientists, but to produce business leaders who can collaborate with them effectively: people who can ask the right questions, spot the right opportunities, and recognize when a model's outputs should—or should not—be trusted.

These notes are a work in progress. Sections continue to be revised, expanded, and reorganized based on classroom experience. Suggestions and corrections are welcome.

George Georgiadis

March 2026

In loving memory of my father,

Iordanis Georgiadis

(1949–2026)

Contents

Part II Unsupervised Learning

Part III Reinforcement Learning

Part I

Supervised Learning

1 Introduction to Supervised Learning

Supervised learning is the workhorse of modern machine learning. The idea is simple: we use **labeled data**—past examples where the outcome is known—to learn a rule that predicts outcomes for new, unseen cases.

Each labeled example consists of a set of **input features**, denoted $\mathbf{x}$, which is information we know at prediction time, and an **outcome** or **label**, denoted y, which is what we want to predict.

A supervised learning algorithm uses many such examples to learn a function that maps inputs to outputs. The end product is typically a *score* (e.g., a predicted probability that a customer will default) or a *forecast* (e.g., next quarter sales), which can then be used to support decisions.

Students seeking additional depth may consult James et al. (2023), Hastie, Tibshirani and Friedman (2009), or Bishop (2006).

1.1 A Concrete Example: Customer Churn

To fix ideas, consider predicting *customer churn*—whether a customer will stop doing business with a firm. Many companies use supervised learning to identify at-risk customers so they can intervene (e.g., retention outreach, targeted offers, service improvements).

Suppose we have historical data on past customers. For each customer we observe features (demographics, usage patterns, billing method, tenure, etc.) and we also know the outcome: did the customer churn? Table 1 shows an example of what such a labeled dataset might look like.

A model trained on data like Table 1 learns patterns that historically preceded churn. The goal is not to "explain churn" in a causal sense, but to *predict* churn well enough to support action.

Customer#	Months w/ firm	Avg. Monthly Charge	Autopay	Churned?
1	2	\$39.95	No	Yes
2	12	\$44.65	Yes	No
3	5	\$49.95	No	Yes
4	36	\$36.69	Yes	No
5	24	\$34.38	Yes	No

Table 1: Example of a labeled dataset for customer churn prediction. Each row is one customer with illustrative features $\mathbf{x}$ and a label y indicating whether they churned (Yes/No).

Prediction vs. Causation. Supervised learning is primarily about *prediction*: given what we know now, what is likely to happen next? This differs from traditional econometric modeling (e.g., regression analysis taught in DECS-431), which is often used for *inference* and causal interpretation under explicit assumptions. Machine learning models can be very accurate predictors, but high predictive accuracy does *not* automatically imply that changing a feature will change the outcome. If the goal is to estimate the effect of an intervention (e.g., "Will offering a discount reduce churn?"), we typically need experimental or quasi-experimental methods in addition to prediction models (Breiman, 2001*b*).

1.2 From Predictions to Decisions

The model output is usually an *input to a decision rule*, not the final decision itself. For example, a churn model might output a predicted probability that a customer with given features will churn; i.e., $\Pr(\text{churn} \mid \mathbf{x}_i)$. The firm must then choose:

- a **threshold** (i.e., who to target), possibly different by segment;
- an **intervention policy** (i.e., what action to take for a given risk score); and
- a **budget constraint** (i.e., how many customers can be contacted and what offers are affordable).

This is why evaluation should reflect business costs and benefits, not just statistical fit.

1.3 Classification vs. Regression Tasks

Supervised learning problems are commonly grouped by the type of outcome y:

- **Classification:** y is a discrete category (e.g., churn Yes/No; risk tier A/B/C).
- **Regression:** y is a continuous number (e.g., price, revenue, demand, lifetime value).

The output type drives how we measure success. For classification we often start with accuracy, but in many settings classes are imbalanced and mistakes have different costs (e.g., missing fraud vs. flagging a legitimate transaction). In such cases, metrics like *precision/recall*, *ROC/AUC*, and cost-based thresholding are more appropriate. For regression we care about how close predicted values are to true values, using metrics like Mean Absolute Error (*MAE*) or Mean Squared Error (*MSE*).

Examples of classification tasks: Approve vs. deny a loan (based on applicant data), fraudulent vs. legitimate transactions, Customer churn (leave vs. stay), spam vs. not spam, image labeling (e.g., object category).

Examples of regression tasks: House price prediction from property features, sales or demand forecasting, customer lifetime value estimation, lead time or delivery time prediction.

1.4 Running Examples

We will return to two examples as we introduce different supervised learning methods:

Classification Example—Customer Churn. As discussed above, predicting customer churn is a prime example of a classification problem. Using a dataset like the one in Table 1, a model's task is to classify each customer as either likely to churn or not. Churn prediction is valuable because it helps companies identify customers who are at risk of

leaving and then take proactive steps to retain them. We will revisit the churn example to see how different algorithms (such as logistic regression, decision trees, and ensemble methods) can be applied to improve churn predictions.

Regression Example—Real-estate Valuation. For a regression-oriented example, we will consider predicting real estate prices. Here the model's job is to estimate a continuous value: the selling price of a house given features such as the house's size, location, age, and so on. This scenario will help illustrate how regression algorithms (like linear regression, k-nearest neighbors, etc.) fit a function to predict numeric outcomes.

1.5 Notation

Throughout these notes we use a small set of notational conventions. Additional notation (dot products, distance, loss minimization, and so on) will be introduced when it is first needed.

Scalars, vectors, and matrices. Individual numerical values (scalars) are written in lower-case, such as z, x, or y. When we collect several numbers into a single object—for example, a list of features—we use boldface lower-case letters to denote a *vector*, such as $\mathbf{z}$, $\mathbf{x}$, or $\mathbf{y}$. A matrix (an array of numbers arranged in rows and columns) is denoted by an upper-case letter, such as Z, X, or Y.

Features and outcomes. We write $\mathbf{x}$ (bold) to denote the vector of p features for an observation:

$$\mathbf{x} = (x_1, x_2, \ldots, x_p),$$

and y to denote the outcome we want to predict.

Indexing observations. With N observations, we index observations by $i = 1, \ldots, N$ and features by $j = 1, \ldots, p$. Thus, $\mathbf{x}_i$ is the feature vector for observation i, x_{ij} is the value of feature j in observation i, and y_i is the outcome for observation i.

Predictions. A "hat" on top of a symbol indicates an estimate or prediction: $\hat{y}_i$ is the model's prediction for observation i, while y_i is the true observed outcome.

1.6 A Typical Supervised Learning Workflow

Most supervised learning projects follow a repeating workflow:

i. **Define the prediction target.** What exactly is y (and over what horizon)? What is the unit of prediction (customer, customer-month, transaction)?

ii. **Assemble labeled data.** Collect features that are available at prediction time and reliable labels.

iii. **Split the data.** Create training/validation/test sets avoiding leakage.

iv. **Preprocess and engineer features.** Clean data, encode categories, scale where appropriate, and build domain-informed features.

v. **Train model(s) and tune hyperparameters.** Fit candidate models on the training set; choose settings using the validation set (sometimes via cross-validation).

vi. **Evaluate.** Use the test set for a final estimate of performance; translate metrics into business impact.

vii. **Deploy and monitor.** Track performance over time and retrain when needed.

1.7 Data Preprocessing

Before any model can be trained, the raw data must be cleaned and transformed into a structured, numeric format—a process called **data preprocessing**. This involves re-

moving duplicates and correcting errors, handling missing values, converting categorical variables (such as city names or contract types) into numbers, and scaling numeric features so that no single variable dominates the model simply because of its units. Crucially, the order in which these steps are carried out matters: the data must be split into training, validation, and test sets *before* any transformation whose parameters are estimated from data (such as computing a column average for imputation or a standard deviation for scaling), so that the test set remains truly unseen. Section 2 covers each of these steps in detail.

1.8 Training a Model

Training a supervised learning model means learning a mapping from inputs to outputs. Conceptually, we assume there is some unknown relationship between input features and outcomes, that is, there is an unknown true function f^* such that

$$y = f^*(\mathbf{x}) + (\text{random noise}).$$

A supervised learning algorithm tries to learn a function f that approximates the true function f^* from the sample data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$.

Different algorithms embody different *inductive biases*: built-in assumptions about what kinds of relationships are likely. Linear regression assumes roughly additive linear effects; decision trees assume outcomes can be explained by a sequence of if-then splits; neural networks allow highly flexible nonlinear functions.

Underfitting vs. Overfitting. A model that is too simple *underfits*, that is, it misses real structure. A model that is too flexible *overfits*; i.e., it learns noise and quirks that do not repeat. The goal is to find a model that captures stable patterns while ignoring random noise (Geman, Bienenstock and Doursat, 1992).

Figure 1.1 illustrates this tension in the context of default risk by credit score. Each point shows, for given credit score, the fraction of borrowers who experienced a default. The true relationship is S-shaped (and noisy). If we fit a simple model, say a linear one (left panel), it will be too flat to follow the curvature of the data, resulting in large errors (even on the training points). By contrast, a very flexible model (middle panel) will wiggle to chase random noise in the training sample yielding low training error, but this comes at a cost of poor predictions on new borrowers. The right panel strikes a middle ground: it is flexible enough to capture the main curvature while ignoring noise.

Figure 1.1: A simple regression problem illustrating underfitting, overfitting, and a good balance.

Hyperparameters and Cross-Validation. Most models have *hyperparameters* (settings chosen outside the training procedure), such as tree depth, regularization strength, or k in k nearest neighbors. We typically select hyperparameters using validation data, sometimes via cross-validation, and then report final performance on the test set.

For instance, in 5-fold cross-validation, the dataset is randomly divided into 5 equal parts (folds). We train the model on 4 of the parts and test it on the remaining part, and then repeat this process 5 times so that each part serves as the test set once. We then average the performance across the 5 runs to obtain an overall evaluation score. Cross-validation thus provides a more robust assessment by reducing the chance that our evaluation is skewed by any particular choice of training/test split (Stone, 1974).

High Dimensionality. When there are many features relative to observations, flexible models can memorize the training data rather than learn a general pattern. In such situations, the model might perform extremely well on the training data but fail to generalize to new data—a symptom of overfitting. Common techniques to mitigate overfitting include feature selection, regularization (which penalizes overly complex models), dimensionality reduction (which involves combining features), and careful validation.

1.9 Evaluating a Model

Once we have trained a model, we must evaluate whether it will *generalize* (i.e., perform well on new, unseen data) and whether it will improve decisions, and therefore create value. In practice, this means looking beyond "Does it fit the training set?" and asking "Will it hold up out-of-sample, and are the mistakes the model makes acceptable given the business costs?"

A Baseline Matters. Any model should first be compared against a simple benchmark. For classification, a natural baseline is to always predict the majority class. For example, in the customer churn setting, if 75% of customers do not churn, a model that always predicts "no churn" will achieve 75% accuracy—without learning anything useful. For forecasting, a common baseline is a "naive" rule such as predicting last quarter's sales again. If a more sophisticated model only marginally improves on these benchmarks, it may not be worth the added complexity, implementation effort, or operational risk.

Classification Models. Binary classification models (e.g., logistic regression for a yes/no outcome) often output a score or probability $\widehat{p}(\mathbf{x}) \in [0, 1]$, interpreted as the predicted probability that $y = 1$ given features $\mathbf{x}$. To turn this score into an action, we choose a *decision threshold*. A common default is 0.5 (predict "positive" if $\widehat{p}(\mathbf{x}) \geq 0.5$), but in business settings the best threshold typically depends on the relative costs of false positives versus

false negatives.

A simple way to formalize this trade-off is to assign a cost c_{FP} to a false positive (model predicts 1 when the truth is 0) and a cost c_{FN} to a false negative (model predicts 0 when the truth is 1). The cost-minimizing rule is to predict "positive" whenever $\widehat{p}(\mathbf{x}) \geq \theta$, where the threshold

$$\theta = \frac{c_{FP}}{c_{FP} + c_{FN}}.$$

For example, in fraud detection, missing fraud is costly (say, $c_{FN} = \$50$) whereas investigating an alert is relatively inexpensive (say, $c_{FP} = \$5$). Therefore, $\theta = 5/(5+50) \simeq 0.09$, which means that a transaction should be flagged even if the model assigns only about a 9% chance of fraud, because missing fraud is far more costly than investigating an extra case.

Performance is often summarized by a confusion matrix (Table 2), which tallies predictions vs. actual outcomes in terms of True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN):

	Predicted: Positive	**Predicted: Negative**
Actual: Positive	TP (true positive)	FN (false negative)
Actual: Negative	FP (false positive)	TN (true negative)

Table 2: Confusion matrix for binary classification.

From these four counts, several useful metrics can be defined:

- **Accuracy** is the proportion of all examples correctly classified:

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

  For example, if a classifier labels 92 out of 100 instances correctly, its accuracy is 92%. Note however, accuracy can be misleading when classes are imbalanced; e.g., if 97% of transactions are legitimate, a model that achieves 98% accuracy is hardly better than marking all transactions as legitimate.

- **Precision** (positive predictive value) is:

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Precision answers: "When the model flags a case as positive, how often is it right?" High precision means few false alarms.

- **Recall** (true positive rate, or sensitivity) is:

$$R = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Recall answers: "Of all true positive cases, how many did we catch?" High recall means few missed positives.

- **F1-score** is the harmonic mean of precision and recall:

$$F1 = \frac{2PR}{P + R}.$$

It is high only when *both* precision and recall are high, and is often used when positives are rare and both types of errors matter.

- **ROC Curve** plots the True Positive Rate (TPR, i.e., Recall) against the False Positive Rate (FPR),

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \qquad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}},$$

as the classification threshold θ varies. Intuitively, θ controls how willing we are to call something "positive" given a model score $\widehat{p}(\mathbf{x})$. If we *lower* θ, we label more cases as positive. That typically increases both TP and FP: we catch more true positives (TPR rises), but we also generate more false alarms (FPR rises). At the extreme, if $\theta = 1$ we almost never predict positive, so we get $(\text{FPR}, \text{TPR}) = (0, 0)$. If $\theta = 0$ we predict everything as positive, so we end up at $(1, 1)$. The ROC curve shows the

entire trade-off between catching positives and triggering false alarms as we sweep θ.

A curve closer to the *upper-left* corner is better because the upper-left means *high TPR* (we catch most true positives) while keeping *low FPR* (few false alarms). In other words, the model separates positives from negatives well: we can choose a threshold that captures many positives without paying too much in false positives.

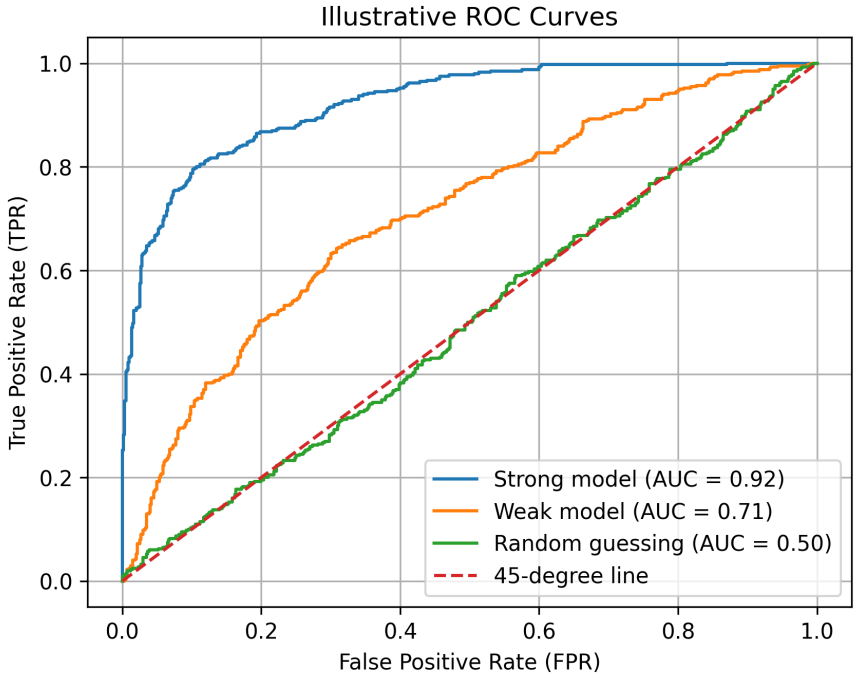


Figure 1.2: Illustrative ROC curves for three classifiers. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) as the decision threshold varies. A stronger model bends toward the upper-left, indicating high TPR at low FPR (better separation of positives from negatives), while random guessing lies near the 45-degree line with AUC ≈ 0.5.

AUC (Area Under the ROC Curve) is a single-number summary of the ROC curve. In binary classification, AUC $= 0.5$ corresponds to random guessing and AUC $= 1$ to perfect ranking. A useful interpretation is: AUC is the probability that the model assigns a higher score to a randomly chosen positive instance than to a randomly chosen negative instance. So higher AUC means the model does a better job *ranking* cases (positives tend to receive higher scores than negatives), independent of any

single threshold choice (Fawcett, 2006).

- **Precision–Recall (PR) Curve** plots precision against recall as the classification threshold θ varies. When we lower θ, we label more cases as positive: recall rises (we catch more true positives), but precision typically falls (a larger share of our "positive" predictions turn out to be wrong). The PR curve traces this trade-off. A model whose PR curve stays close to the *upper-right* corner—high precision *and* high recall simultaneously—is performing well. Just as AUC summarizes the ROC curve, the **Average Precision (AP)** score summarizes the PR curve in a single number; higher is better.

PR curves and ROC curves both summarize classifier performance across thresholds, but they are *not* interchangeable. The key difference is how they behave under **class imbalance**—when one class is far more common than the other. For example, in fraud detection there are vastly more legitimate transactions than fraudulent ones. The ROC curve's x-axis is the false positive rate, $\text{FPR} = \text{FP}/(\text{FP} + \text{TN})$. When TN is enormous, even a large number of false positives translates into a small rate—so the ROC curve can look reassuringly close to the upper-left corner even when the model generates many false alarms in absolute terms. Precision, by contrast, is $\text{TP}/(\text{TP} + \text{FP})$: it is directly hurt by every false positive, regardless of how many true negatives exist. A PR curve will therefore reveal weaknesses that the ROC curve can mask. The rule of thumb is to use ROC curves (and AUC) when the classes are roughly balanced, and to use PR curves (and AP) when the positive class is rare and the business cost of false positives matters.

Regression Models. For models like linear regression that predict continuous outcomes, we evaluate the magnitude of the prediction error:

- **Mean Squared Error (MSE)** averages squared errors:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (\widehat{y}_i - y_i)^2 .$$

Squaring penalizes large mistakes heavily, which can be desirable when big errors are especially costly.

- **Mean Absolute Error (MAE)** averages absolute errors:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N} |\widehat{y}_i - y_i| \,.$$

Absolute value, written $|u|$, measures magnitude without regard to sign.

MAE is often more robust to outliers because it does not disproportionately amplify large errors. It can be read as "the typical absolute miss" in the units of y.

A related metric is the **Mean Absolute Percentage Error (MAPE)**,

$$\text{MAPE} = \frac{1}{N}\sum_{i=1}^{N} \frac{|\widehat{y}_i - y_i|}{|y_i|} \times 100\%.$$

MAPE expresses the typical miss as a *percentage* of the true value rather than in raw units, which makes it useful when stakeholders think in relative terms ("our forecasts are off by about 8%") or when comparing accuracy across items of very different magnitudes—for example, forecasting demand for both a $5 accessory and a $500 appliance. Its main limitation is that the denominator is the actual value y_i, so MAPE becomes unstable when y_i is near zero and is undefined when $y_i = 0$. For this reason, it is best used when the target is strictly positive and bounded away from zero.

- **R-squared (R^2)** measures explained variance relative to predicting the mean:

$$R^2 = 1 - \frac{\sum_i (y_i - \widehat{y}_i)^2}{\sum_i (y_i - \overline{y})^2},$$

where the "overbar" $\bar{y}$ denotes the sample average: $\bar{y} = (1/N)\sum_{i=1}^{N} y_i$.

An R^2 of 0 means the model is no better than always predicting $\overline{y}$; an R^2 of 0.75

means the model explains 75% of the variance in y. A high R^2 does *not* by itself guarantee a good model—we still need to check out-of-sample performance and watch for overfitting.

In the following sections we catalog major supervised learning paradigms (regression, decision trees, k nearest neighbors (kNN), support vector machines (SVMs), neural networks, and more). For each, we discuss how the method works, when it tends to perform well, common pitfalls, and best practices.

1.10 Dealing with Class Imbalance

In many classification problems that matter to businesses, the event of interest is rare. Credit-card fraud affects under 1% of transactions. Customer churn in a healthy subscription business might run 3–5% per month. Loan defaults may be 2–3% of the portfolio. When one class vastly outnumbers the other, we say the dataset is **class-imbalanced**.

Why imbalance is a problem. As noted in Section 1.9, accuracy can be deeply misleading when classes are imbalanced. If only 1% of transactions are fraudulent, a model that *never* flags fraud achieves 99% accuracy—while catching zero fraud. The business cost of that model is enormous. The core issue is that a standard training procedure optimizes overall accuracy and may learn to ignore the minority class because doing so barely hurts the overall error rate.

Strategies for handling imbalance. Four practical approaches are commonly used, often in combination:

1. **Class-weighted loss functions.** Instead of treating every prediction error equally, we tell the model that errors on the minority class are more costly. Most ML libraries allow you to set *class weights*: for example, giving the "fraud" class a weight of 50 and the "legitimate" class a weight of 1. This makes the training procedure pay

50× more attention to missed fraud cases. Conceptually, this is similar to the cost-based thresholding discussed in Section 1.9, but it operates *during training* rather than after. Class weighting is the simplest and often most effective first step.

2. **Oversampling the minority class.** We duplicate (or create synthetic copies of) the rare-class examples so that the training set is more balanced. The most widely used technique is **SMOTE** (Synthetic Minority Over-sampling Technique), which creates new minority-class examples by interpolating between existing ones rather than simply duplicating them (Chawla et al., 2002). Oversampling is applied *only to the training set*—never to the validation or test set, which must reflect the real-world class distribution.

3. **Undersampling the majority class.** We randomly remove some majority-class examples to reduce the imbalance. This is simple and fast, but it discards potentially useful data. It works best when the majority class is very large and the model can afford to lose examples.

4. **Threshold tuning.** Even without rebalancing the data, we can adjust the decision threshold (Section 1.9) to reflect the business costs of false positives versus false negatives. For fraud detection, lowering the threshold so that we flag transactions with even a small probability of fraud is often the right approach.

Practical guidance. Start with *class-weighted training* (approach 1), because it requires no data manipulation and is supported by virtually every ML library. If performance on the minority class is still insufficient, add SMOTE or undersampling and compare results on the *validation set*. Always evaluate using metrics that account for imbalance—precision, recall, F1, and AUC—rather than raw accuracy. Finally, remember that resampling strategies change the *training set only*; the validation and test sets should always mirror the true class proportions so that evaluation reflects real-world performance.

1.11 A Preview: Interpretability and Model Selection

Before we turn to specific algorithms, it is worth flagging a theme that will recur throughout these notes: *not all models are equally easy to explain*.

A linear regression model produces a set of coefficients that can be read as "each extra square foot adds $200." A small decision tree can be printed as a flowchart and walked through step by step. These are **interpretable** models: a manager, a regulator, or a customer can understand *why* the model made a particular prediction.

Other models—large ensembles of hundreds of trees, or deep neural networks with millions of parameters—can be substantially more accurate, but their reasoning is opaque. They are sometimes called **black-box** models. You can see the inputs and the output, but explaining exactly how the model combined hundreds of features to arrive at a particular score is difficult.

This distinction matters for at least three reasons:

1. **Managerial adoption.** A model that cannot be explained is harder to trust and harder to act on. If a churn model flags a customer as high risk, the retention team will want to know *why*—is it billing issues, support complaints, or contract type? An interpretable model answers that question directly.

2. **Regulatory and governance requirements.** In domains such as lending, insurance, and healthcare, regulations may require that decisions be explainable. A bank that denies a loan application may be legally required to state the reasons.

3. **Debugging and trust.** When an interpretable model makes a surprising prediction, you can inspect the reason and determine whether the model is using a sensible signal or exploiting a data artifact. With a black-box model, this is harder.

Fortunately, the choice is not all-or-nothing. Interpretability tools such as SHAP values, partial-dependence plots, and counterfactual explanations can shed light on black-

box models, making them more transparent without sacrificing accuracy (Molnar, 2022). We cover these tools in detail in Section 10.1.

As we introduce each supervised learning paradigm in the sections that follow, we will note where it falls on the **interpretability–flexibility spectrum**: simpler models tend to be more interpretable but less flexible, while more complex models can capture richer patterns but are harder to explain. Choosing a model is ultimately about finding the right balance for the business context at hand.

2 Data Preprocessing

Data preprocessing refers to the steps taken to convert raw, messy data into a clean, structured format that can be fed into a machine learning model. In practice, data quality is often the limiting factor: training a sophisticated model on flawed data produces (sophisticated) nonsense.

A critical principle governs the order in which these steps are carried out: *any transformation whose parameters are estimated from data*—such as computing a column mean for imputation, defining category mappings for encoding, or computing a min and max for scaling—*must be estimated from the training set only, after the data has been split*. Estimating these parameters on the full dataset (including the validation and test sets) is a form of **data leakage**: information from the test set may quietly influence training, making the model appear better than it will perform in practice.[1]

Figure 2.1 illustrates the resulting two-phase pipeline. In the first phase, we perform cleaning operations that do not require learning anything from the data (removing duplicates, fixing errors, standardizing formats). We then split the data. In the second phase, we fit all data-dependent transformations—imputation, encoding, and scaling—on the training set and apply those fitted transformations to the validation and test sets without refitting.

2.1 Data Cleaning

The first step is typically **data cleaning**: detecting and correcting problems in the dataset. The operations below do not require estimating statistics from the data, so they can safely be performed *before* splitting.

- **Duplicate records.** If the same customer or transaction appears more than once, the

[1] Think of it this way: a model seeing the test data during training is like a student previewing exam questions before they study—their preparation looks better than it really is, but they have not actually learned more.

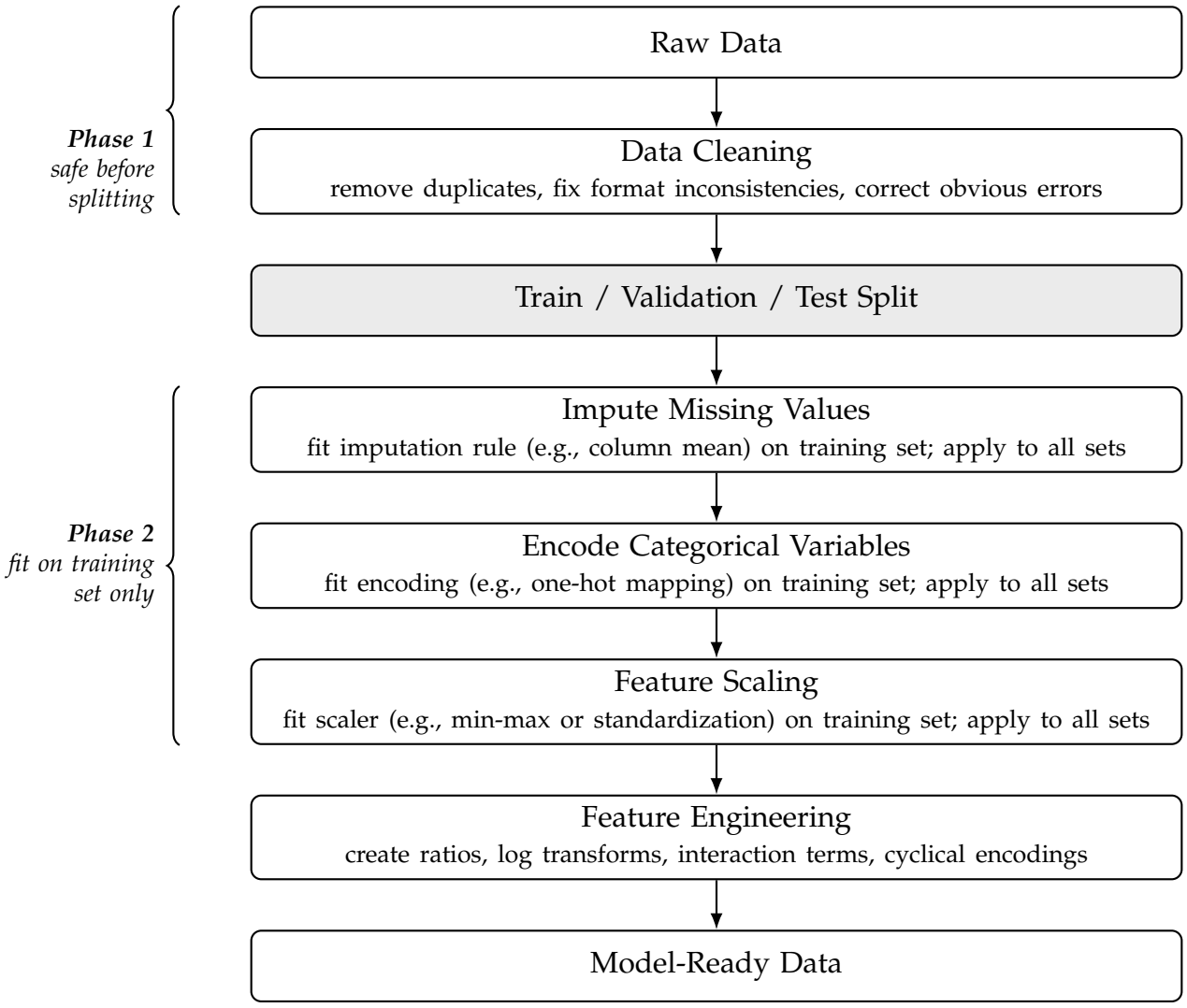


Figure 2.1: A data preprocessing pipeline. **Phase 1** operations do not learn parameters from the data and can be performed before splitting. The data is then split into training, validation, and test sets. **Phase 2** operations involve estimating parameters from data; these must be fit on the training set and then applied to the validation and test sets without refitting, to avoid data leakage. Feature engineering can occur at various points depending on whether the transformation requires statistics from the data.

duplicates should be identified and removed so they do not artificially inflate the model's confidence in those patterns.

- **Inconsistent formats.** Dates should all follow the same format, monetary values should use the same currency and unit, and categories should be standardized—for example, "NY" and "New York" should be mapped to the same label.

- **Obvious errors.** A recorded age of 900 years or a negative price is almost certainly a data-entry mistake. Such values should be corrected (if the true value can be determined) or removed.

2.2 Handling Outliers

Outliers are data points far outside the normal range; for example, a house price ten times higher than the next most expensive house, or a loan amount recorded in cents instead of dollars (making it 100× larger than intended). A few extreme values can distort models that rely on averages or distances, pulling predictions toward the outlier. The appropriate remedy depends on *why* the outlier exists:

- If the value is clearly a **data-entry error** (e.g., a $3,000 monthly charge for a $30 plan), correct it if possible or remove the record. This can safely be done before splitting.

- If the value is **real but extreme** (e.g., a genuinely ultra-expensive property), consider *capping* (also called *winsorizing*); i.e., replacing values beyond a chosen percentile (say, the 99th) with the value at that percentile. Because computing percentiles requires estimating statistics from data, capping should be fit on the training set and applied to all sets.[2]

[2] In practice, many analysts cap before splitting when the percentile thresholds are obvious domain knowledge (e.g., "no house costs more than $50M"). The strict rule is to fit on training data; the pragmatic rule is to be aware of the risk and document the choice.

- If extreme values are a natural feature of the domain (e.g., income distributions are highly skewed), a **log transformation** of the variable can compress the range and reduce the influence of large values. A log transform does not depend on the data's statistics (it is the same function regardless of the dataset), so it is safe before splitting.

2.3 Splitting Data for Training, Validation, and Testing

After basic cleaning, and *before* any transformation that requires learning from the data, we split the dataset into three parts:

- a **training set** to fit model parameters and preprocessing rules;
- a **validation set** to choose models and hyperparameters; and
- a **test set** for a final, unbiased evaluation.

Typically, the training set receives the majority of the data (say, 60–70%), with the remainder divided between validation (15–20%) and test (15–20%). This practice helps us gauge whether the model is truly learning general patterns or just memorizing the quirks of the training data.

Why split before imputing, encoding, and scaling? Each of these steps requires computing something from the data: a column mean (for imputation), a set of category labels (for encoding), or a min/max or standard deviation (for scaling). If we compute these on the full dataset, the training set implicitly "sees" information from the test set. This is **data leakage**—it inflates apparent performance during development but causes the model to underperform in production.

Splitting by entity or by time. A random split is appropriate when observations are independent. Two common exceptions require more care:

1. **Multiple records per entity.** If the same customer, patient, or store appears in multiple rows, split *by entity* so that all records for a given customer land in the same partition. Otherwise, the model can partly memorize entity-specific patterns and appear to generalize when it has not.

2. **Time-series data.** If the task involves predicting the future, split *by time*: train on earlier periods, validate and test on later periods. A random split would let the model "peek" at future data.

2.4 Handling Missing Values

Missing data is extremely common. A loan application might lack a customer's income; a house listing might omit the year built. Missing values can distort analysis and introduce bias if not handled properly (and crash training if not handled at all). Because most imputation methods require computing summary statistics (column means, medians, or subgroup averages), *imputation must be fit on the training set and applied to all sets*.

Three standard approaches are:

1. *Drop the record* entirely, which is reasonable when few records are affected and the missingness appears random. Dropping does not require fitting anything, so it can be done before or after splitting.

2. *Impute* (fill in) the missing value with a reasonable estimate, such as the column's mean or median, or the mean within a subgroup (e.g., the average house age in the same zip code). The mean or median **must be computed from the training set**, then used to fill missing values in the validation and test sets as well.

3. *Add a "missing" indicator.* Create a new binary feature that flags whether the value was originally missing, then impute as above. This lets the model learn whether the fact that data was missing is itself informative; e.g., customers who decline to provide income information may behave differently from those who do.

A key concern is *systematic missingness*: if the data is missing for a reason related to the outcome (e.g., high-risk borrowers are less likely to report income), naive imputation can bias the model. When missingness may be systematic, it is especially important to add a missing indicator and to inspect how model performance changes under different imputation strategies.

2.5 Encoding Categorical Variables

Many datasets contain **categorical variables**: non-numeric information such as customer location, product category, contract type, or industry sector. Most machine learning algorithms require numeric inputs, so categories must be converted into numbers. Because the encoding defines a mapping (which categories exist, and how they are numbered), *the mapping should be determined from the training set* and then applied consistently to validation and test data.

One-hot encoding (dummy variables). The most common approach is *one-hot encoding*: create a separate binary (0/1) column for each category. For example, if a house-price dataset has a "Location" column with values "New York," "San Francisco," and "Chicago," one-hot encoding produces three new columns—one per city. A house in New York has Location_NewYork $= 1$ and the other two columns equal to zero.

When a categorical variable has many rare categories (e.g., hundreds of ZIP codes), it is common to group infrequent values into an "Other" category to prevent the model from seeing each rare category only once or twice. If a category appears in the test set but was never seen during training, it is typically mapped to "Other" as well.

Ordinal encoding. If a categorical variable has a natural ordering (e.g., satisfaction ratings Low / Medium / High), we can map the categories to integers (1, 2, 3) that respect that order. This is appropriate only when the ordering is meaningful; it would be wrong

to impose an order on, say, city names.

Text data. Free-form text (e.g., a real-estate listing description or a customer-service note) requires a different treatment. Simpler methods include *bag-of-words*, which counts how often each word appears. More modern methods use *text embeddings*: pre-trained models that map a block of text into a numeric vector designed to capture meaning.[3]

2.6 Feature Scaling

Feature scaling transforms numeric values onto a common scale. This matters because raw features often have very different ranges. For example, a credit-risk dataset might contain a borrower's age (18–90) alongside annual income ($20,000 to millions). Without scaling, a model that relies on distances or weighted sums will be dominated by the feature with the largest numeric values—income would swamp age, even if age is equally important.

Feature scaling is essential for models that use distances or gradient-based optimization, including k-nearest neighbors, support vector machines, neural networks, and regularized linear models. It is less important for tree-based models (decision trees, random forests, gradient boosting), which make decisions based on thresholds rather than numeric magnitudes.

Common scaling methods. The two most widely used techniques are:

- **Min–max normalization** which rescales each feature to the range $[0, 1]$:

$$x_{ij}^{\text{scaled}} = \frac{x_{ij} - \min_l\{x_{lj}\}}{\max_l\{x_{lj}\} - \min_l\{x_{lj}\}}.$$

[3]Conceptually, text-embedding models try to ensure that two blocks of text that are semantically similar are close to each other in vector space, and vice versa.

For instance, if house sizes in the *training set* range from 800 to 5,000 square feet, then 800 maps to 0.0 and 5,000 maps to 1.0.

- **Standardization** (also called z-score scaling) which centers each feature at 0 with a standard deviation of 1:

$$x_{ij}^{\text{scaled}} = \frac{x_{ij} - \text{mean}_l\{x_{lj}\}}{\text{std}_l\{x_{lj}\}}.$$

  Standardization is the usual default because it handles outliers more gracefully than min–max (a single extreme value does not compress the rest of the data into a narrow band).

In both formulas, the statistics (min, max, mean, std) are **computed from the training set only**.

A critical operational point. The scaling rule is learned from the training data; e.g., the training set's mean and standard deviation. When scoring new instances at prediction time—whether on the validation set, the test set, or live production data—you must apply the *same* transformation using the training set's parameters, not recalculate new ones.

Scaling the target variable. Everything above concerns scaling the *input features*. For **regression** tasks, you may also choose to transform the target y—for instance, by taking the logarithm of house prices to reduce skewness, or by standardizing revenue figures. If you do transform y before training, the model's raw output will be in the transformed scale. To get predictions in original units (dollars, units sold, etc.), you must reverse the transformation after prediction; e.g., exponentiate if you took the log. For **classification**, the target is already a category label (e.g., churn / no churn) and is not scaled.

2.7 Feature Engineering

Feature engineering is the process of transforming and combining existing variables to create new inputs that better capture the patterns in the data. A well-engineered feature can dramatically improve a model's performance—not by making the algorithm more complex, but by presenting the information in a form the algorithm can use more effectively. Feature engineering is often the area where *domain knowledge* has the biggest payoff: understanding the business context helps you create features that reflect how the world actually works.

Ratios and relative measures. Raw numbers are sometimes less informative than the ratios between them. In finance, the debt-to-equity ratio or interest-coverage ratio is more interpretable than raw balances; in operations, utilization (run hours divided by available hours) and on-time delivery rate summarize efficiency better than the raw totals. In a churn model, the ratio of support calls to months of tenure ("calls per month") might be more predictive than either variable alone, because a customer with 10 calls in 2 months is very different from one with 10 calls in 5 years.

Interactions. Sometimes the effect of one variable depends on the value of another. For example, a borrower's income and credit score together reflect repayment capacity better than either alone: a high income matters more if the credit score is also good. Creating an *interaction feature*—e.g., income $\times$ credit score—lets the model capture this joint effect. Similarly, in a house-price model, the value of additional living space may depend on the lot size: an extra 500 square feet is worth more on a large lot than on a tiny one. Creating the feature (lot sq.ft) $\times$ (house sq.ft) allows the model to learn this relationship.

In practice, it is better to add a small number of meaningful interactions suggested by the business setting than to generate all possible pairs indiscriminately—the latter creates a very large number of features and increases the risk of overfitting (Kuhn and Johnson,

2019).

Nonlinear transformations. Many real-world relationships are not straight lines. For instance, square footage may have a *diminishing* effect on the value of a home: an extra 500 sq ft has a larger effect on a 1,000 sq ft home than on a 5,000 sq ft home. Common transformations include:

- $\log(x)$ — compresses large values and captures diminishing returns. Widely used for prices, incomes, and other right-skewed variables.
- $\sqrt{x}$ — a milder compression than the log; useful when the effect tapers but does not flatten entirely.
- x^2 — captures *accelerating* effects; for example, the cost of delay might increase quadratically with wait time.

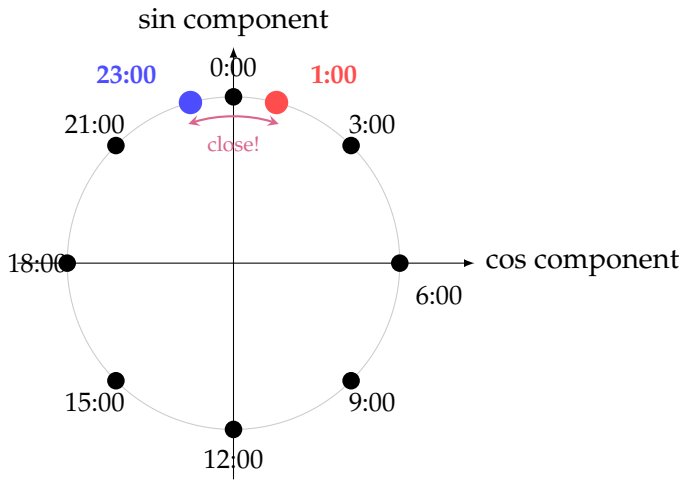
After adding a transformed feature, the model remains linear *in the transformed inputs* (e.g., a linear regression on $\log(\text{sq ft})$ is still a linear model), which preserves the advantages of simplicity and interpretability.

Temporal features. When outcomes evolve over time, features that summarize recent history can be very powerful:

- **Lags** reuse prior values as features. For example, last month's sales, yesterday's stock price, or the customer's balance 90 days ago. Lags allow the model to learn from recent trends and shocks.
- **Rolling windows** summarize short-term patterns. A 7-day moving average of daily sales smooths out day-to-day noise and reveals underlying trends; a 4-week rolling standard deviation captures recent volatility. In a churn model, computing a customer's average monthly bill over the past three months can reveal recent changes in spending patterns.

Cyclical features. Some time variables are *cyclical*: after 23:00 comes 00:00; after December comes January. Treating hours or months as ordinary integers is misleading because it places the endpoints far apart numerically—hour 0 and hour 23 would be 23 units apart, even though they are adjacent in time.

A simple fix is to represent each cyclical variable as a position on a circle, using a pair of coordinates.[4] Figure 2.2 illustrates this idea for hours of the day. The key intuition is that after the mapping, times that are *close on the clock* are also *close in feature space*, regardless of whether they straddle midnight. Any standard software library can generate these features automatically.



Each hour maps to a point on the circle. Hours that are close in time (e.g., 23:00 and 1:00) are close in the $(\cos, \sin)$ space—even though their integer values (23 vs. 1) are far apart.

Figure 2.2: Cyclical encoding of hour-of-day. Each hour is mapped to a point on the unit circle using a pair of sine and cosine features. Times that are close on the clock face—such as 23:00 and 1:00—are also close in the two-dimensional feature space, which is not the case when hours are treated as plain integers.

Aggregations and group context. In settings with repeated observations per entity, aggregating information at a higher level can create powerful features. For example:

- **Customer-level summaries** for a churn model: average purchase value, number of transactions in the past 90 days, time since last purchase, trend in monthly spending.

[4] Mathematically, if t is the value and T is the period (e.g., $T = 24$ for hours), the two features are $\sin(2\pi t/T)$ and $\cos(2\pi t/T)$. These trace a circle, so midnight and 11 PM end up right next to each other.

- **Contextual aggregates** for a house-price model: neighborhood median price, school-district rating, store-level foot traffic.

These features let the model account for the environment each observation sits in and can filter out noise that is present in individual-transaction-level data.

Feature engineering is ultimately driven by domain knowledge. A good rule of thumb is to add a small number of meaningful transformations suggested by the business setting rather than to generate many indiscriminately—especially because adding too many features increases the risk of overfitting.

2.8 Example: Preprocessing for House Price Prediction

To make the ideas in this section concrete, imagine we have obtained a dataset of roughly 20,000 residential home sales. For each sale, we observe the transaction price and a set of property features: square footage of living space and lot, number of bedrooms and bathrooms, year built, year last renovated (if any), zip code, a construction-quality grade, and the date of sale. Our goal is to build a model that predicts sale price from these features.

Before we can train any model, we need to work through the preprocessing pipeline. Each step below corresponds to a stage in Figure 2.1.

Step 1: Data cleaning (pre-split). We begin with problems that can be identified and fixed without computing any statistics from the data.

- **Exact duplicate rows.** A quick check may reveal rows that are exact copies of another row. These are removed.

- **ID columns.** The dataset includes a property identifier. This should *not* be used as a feature: it carries no predictive information about price and would cause the model to memorize individual properties rather than learn general patterns.

- **Repeat sales.** Some properties appear more than once (which can be inferred from the property IDs) because they were sold multiple times during the sample period. Unlike exact duplicates, these are legitimate records—but they have an important implication for how we split the data later (see Step 2).

- **Date format.** Sale dates are stored as raw timestamps (e.g., `20141013T000000`). We convert each date into a continuous number—for example, 2014.78—that captures both the year and the fraction of the year, which will make it easier to use in a model.

- **Clear data-entry errors.** A handful of records list a year built of 19,430 or similar—clearly a misplaced digit. Dividing by 10 produces a plausible construction year, so we apply that correction. Similarly, some cells contain the string `"err"` instead of a number; we set these to missing so they can be handled during imputation.

- **Structural zeros.** Some houses are recorded as having zero bedrooms or zero bathrooms. These values may reflect missing or miscoded data or non-residential structures (e.g., warehouses, garages, shacks, etc). We flag them as missing so they can be imputed after the split. We might also drop those rows from the dataset if there are not too many of them.

- **Suspect prices.** The most expensive property in the dataset lists at $175 million. To decide whether this (and a handful of other very high prices) reflects a genuine sale or an extra-zero typo, we compute a simple sanity check: price per square foot. If a home's price per square foot is wildly above the 99th percentile, the record is suspicious. We remove or correct the clear errors, leaving genuinely expensive homes in the data for now (we will revisit extreme values via capping or log-transforming below).

Step 2: Train / validation / test split. We split the data into a training set (70%), validation set (15%), and test set (15%). Because some properties were sold more than once, we

split *by property ID*: all sales of the same property go into the same partition. This prevents the model from "memorizing" a property's price from an earlier sale in the training set and then appearing to predict it well in the test set.

Step 3: Impute missing values (fit on training set). After cleaning, some records still have missing entries: the zero-bedroom and zero-bathroom flags from Step 1, the cells that contained `"err"`, and any other originally missing fields. For numeric features we might impute with the *training-set median* of that column (the median is more robust to outliers than the mean). We also create a binary "missing" indicator for each imputed column so that the model can learn whether the fact that data was missing is itself predictive. The same medians and indicators can then be applied to the validation and test sets without recomputing.

Step 4: Encode categorical variables (fit on training set). The zip-code field is categorical: it identifies a neighborhood, but the numeric ZIP values have no meaningful ordering. We apply one-hot encoding, creating a binary column for each zip code observed in the training set. If a zip code appears in the test set but was never seen during training, it is mapped to an "Other" column.

The construction-quality grade (1–13) has a natural ordering—higher grades indicate better construction—so we retain it as an ordinal integer rather than one-hot encoding it.

Step 5: Feature scaling (fit on training set). The numeric features span very different ranges: square footage runs into the thousands, while the number of bathrooms rarely exceeds 5. We standardize each feature using the training-set mean and standard deviation, so that all features are on a comparable scale. The same means and standard deviations are applied to the validation and test sets.

Step 6: Feature engineering. Domain knowledge suggests several useful transformations:

- **Age instead of year built.** Replacing "year built" with "age at time of sale" (sale year minus year built) is more interpretable and avoids the model needing to learn that a higher year means a newer home.

- **Years since renovation.** If a property was never renovated, the "year renovated" field is zero—a value that would confuse most models. We replace it with "years since last renovation," coding never-renovated homes with the home's age or adding a separate "never renovated" indicator.

- **Log-transforming price and living area.** Both sale price and living-space square footage are heavily right-skewed: most values cluster at the low end, with a long tail of expensive or large homes. Taking the logarithm compresses this tail and produces a more symmetric distribution. There is also an economic rationale: housing markets are often well described by a relationship of the form $\text{Price} = C \times (\text{sqft})^{\beta}$, where C is a constant that depends on other features such as location and quality; taking the log of both sides yields $\log(\text{Price}) = \log C + \beta \log(\text{sqft})$, which is linear—exactly the form a linear regression can learn directly.

- **Cyclical encoding of sale date.** The fraction-of-year component of the sale date is cyclical (late December and early January are adjacent in time but far apart numerically). We encode it using sine and cosine features as described in Section 2.7.

After all six steps, every column is numeric, consistently formatted, properly scaled, and free of leakage—and the data is ready for modeling.

3 Linear Regression

Linear regression is one of the simplest and most widely used supervised learning methods for predicting a continuous outcome (James et al., 2023) (Chapters 3 and 6). It assumes that the relationship between the input features $\mathbf{x} = (x_1, x_2, \ldots, x_p)$ and the outcome y is approximately linear. In its basic form (a single-output linear model), it can be written as:

$$\widehat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_p x_p = w_0 + \mathbf{w} \cdot \mathbf{x},$$

where w_0 is the intercept (bias term) and $\mathbf{w} = (w_1, \ldots, w_p)$ is the vector of coefficients (weights) that determine how strongly each feature influences the prediction.

The notation $\mathbf{w} \cdot \mathbf{x}$ is a *dot product*: it means 'multiply each weight by the corresponding feature and add the results'; i.e., $\mathbf{w} \cdot \mathbf{x} = \sum_{j=1}^{p} w_j x_j$. This 'weight-and-add' operation is the basic building block of linear models.

3.1 Training the model

Assume our data comprises N observations: $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, where each $\mathbf{x}_i$ is a vector comprising the p features for observation i, and y_i is the corresponding outcome. The coefficients are estimated from the training data by minimizing a *loss function*—typically, the mean squared error between the predicted and actual outcomes:

$$\min_{w_0, \mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} (y_i - \widehat{y}_i)^2, \tag{1}$$

where $\widehat{y}_i = w_0 + \sum_{j=1}^{p} w_j x_{ij} = w_0 + \mathbf{w} \cdot \mathbf{x}_i$ is the model's prediction for observation i and y_i is the ground truth.[5] This loss function can be quickly minimized using standard methods; e.g., MS Excel, Stata, Python, Matlab, etc.

The symbol $\min_{\mathbf{w}}$ means 'find the weights that make this expression as small as pos-

[5] Recall that x_{ij} denotes the value of feature j in observation i.

sible.' The related notation $\text{argmin}_{\mathbf{w}}$ refers to the specific values of $\mathbf{w}$ that achieve that minimum.

3.2 Example: Predicting house values

We return to the house-price dataset introduced in Section 2.8. After preprocessing, each observation has numeric features. A simple linear regression model might be:

$$\widehat{\text{Price}} = w_0 + w_1 \times (\text{sqft\_living}) + w_2 \times (\text{sqft\_lot}) + w_3 \times (\#\text{bedrooms}) + w_4 \times (\#\text{bathrooms}) + w_5 \times (\text{age}) + w_6 \times (\text{grade}).$$

(We omit zip-code indicators and other features for readability; they enter the model in exactly the same way.) After estimating the coefficients from the training data, we might obtain:

w_0	w_1	w_2	w_3	w_4	w_5	w_6
\$100K	\$200	\$2	−\$50K	\$40K	−\$1K	\$29K

This means that, each home has a baseline predicted value of \$100,000. Then, all else equal, each additional square foot of living space is associated with a \$200 increase in price, each additional square foot of lot adds \$2, each year of age reduces value by about \$1,000, and each one-point increase in construction grade is associated with a \$29,000 premium. Linear regression is popular in part because of this interpretability: each coefficient can be read as the marginal association of that feature with the outcome, holding the other features fixed.

A negative coefficient on bedrooms? Notice that the coefficient on bedrooms is −\$50K. This does *not* mean that adding a bedroom lowers a home's value. Rather, it reflects the fact that bedrooms and living-space square footage are highly correlated: a larger

house typically has more bedrooms. Once we *hold square footage fixed*, a home with more bedrooms is dividing the same space into smaller rooms, which the market prices lower. The coefficient answers a very specific question: "Among homes of the *same size*, same lot, same grade, and same zip code, what is the price difference between those with one additional bedroom?" This is a good reminder that coefficients in a regression model are *conditional* associations—their sign and magnitude can change depending on which other features are in the model.

Prediction vs. causation. More broadly, in a predictive setting these coefficients are associations conditional on the other features in the model; they should not be interpreted as causal effects without additional assumptions or experimental/quasi-experimental design. The fact that *grade* carries a large positive coefficient does not mean that renovating a house to increase its grade will raise its price by exactly $29,000—the coefficient may partly reflect other unobserved factors (e.g., affluent neighborhoods tend to have higher-grade homes).

Multicollinearity. The bedroom example above is a symptom of a broader issue called *multicollinearity*: when two or more features are highly correlated with one another. In our dataset, living-space square footage is strongly correlated with the number of bedrooms, the number of bathrooms, and lot size. Bedrooms and bathrooms are correlated with each other. Grade may be correlated with square footage and age.

Multicollinearity does not, by itself, bias predictions—the model's overall forecasts can still be accurate. What it does is make *individual coefficients unstable*: small changes in the training data (adding or removing a few observations) can cause large swings in the estimated weights. In the extreme, one correlated feature might get a very large positive coefficient while another gets a very large negative coefficient, even though both features are positively associated with price. The two coefficients are "sharing credit" for the same underlying signal, and the split between them is sensitive to noise.

For a manager, the practical implications are:

1. **Be cautious interpreting individual coefficients** when features are correlated. The sign or magnitude of a single weight may not reflect what you expect from domain knowledge, as illustrated by the negative bedroom coefficient.

2. **Predictions are usually fine.** Even when individual coefficients are unstable, the weighted sum $\mathbf{w} \cdot \mathbf{x}$ often remains stable because the shifts in correlated coefficients tend to offset each other.

3. **Regularization helps.** As we discuss in the next subsection, Ridge and Lasso regression explicitly penalize large coefficients, which stabilizes the estimates and reduces the sensitivity to multicollinearity.

Encoding categorical features. Zip codes are categorical: the numeric ZIP values have no meaningful ordering. As discussed during preprocessing (Section 2.5), we one-hot encode them, creating an indicator variable for each zip code. With an intercept in the model, we include $K - 1$ indicators to avoid perfect multicollinearity (one zip code serves as the omitted *baseline*). Each zip-code coefficient then captures the predicted price premium or discount relative to that baseline, holding the other features fixed. For instance, if the baseline is zip code 98001 and the coefficient on 98004 is +\$250K, the model predicts that a home in 98004 sells for \$250,000 more than an otherwise identical home in 98001.

Log specifications. During preprocessing (Section 2.8), we noted that both price and living-space square footage are heavily right-skewed, and that housing markets often follow a relationship of the form $\text{Price} = C \times (\text{sqft})^{\beta}$. Taking logarithms yields:

$$\log(\text{Price}) = \underbrace{\log C}_{\text{intercept}} + \beta \log(\text{sqft\_living}) + \cdots$$

which is linear in the transformed features—exactly what a linear regression can estimate. In this "log–log" specification, the coefficient β has a convenient interpretation: a 1% increase in square footage is associated with roughly a β% increase in price. More generally, when the target is log-transformed, all coefficients are interpreted in percentage terms rather than dollar terms. This is often a better fit for housing data and is the specification we will use going forward.

Limitations: nonlinearity and interactions. Even with the log transformation, the model assumes that each feature has a constant, additive effect on $\log(\text{Price})$. If the true relationship involves effects that change depending on the value of another feature, a linear model will miss them unless the user manually adds *synthetic features*.

For instance, the incremental value of additional living space may depend on the lot size: an extra 500 square feet of living space is worth more on a large lot than on a tiny one. To capture this, we could create an interaction feature $\log(\text{sqft\_living}) \times \text{sqft\_lot}$ with a new coefficient w_7. Similarly, the premium for construction grade may be larger in expensive zip codes than in cheaper ones. Each such interaction requires a deliberate modeling choice.

The model remains linear in its expanded set of features (original and synthetic), so it retains the advantages of simplicity and fast estimation. However, this approach creates two difficulties. First, it is up to the user to decide which interactions and transformations to include; the space of possibilities is vast. Second, adding many synthetic features increases the total number of coefficients, making the model prone to overfitting—especially when features are already correlated with one another, compounding the multicollinearity problem. For example, in a model with p features, including all pairwise interactions requires estimating $p + p(p-1)/2$ coefficients: with $p = 6$ core features, that is already 21 weights; with the dozens of zip-code indicators included, the number grows rapidly.

One systematic way to manage this complexity is *regularization*, which we turn to next.

3.3 Regularization: Controlling model complexity

Regularization is a method for improving the generalization of regression models, particularly when the number of predictors is large or when those predictors are correlated, as with multiplicative interactions. Regularization adds a penalty term to the regression's objective function that discourages large coefficient values; that is, instead of (1), we solve

$$\min_{w_0,\mathbf{w}} \sum_{i=1}^{N} (y_i - \widehat{y}_i)^2 + \lambda \times R(\mathbf{w}),$$

where $\lambda > 0$ is a parameter that controls the strength of the penalty, and $R(\mathbf{w})$ is the penalty function.

The simplest and most common form of regularization is *Ridge regression*, also known as *L2 regularization*, which assumes that

$$R(\mathbf{w}) = \sum_{j=1}^{p} w_j^2.$$

This term penalizes large weights, and the larger λ is, the more the weights are forced towards 0, reducing model complexity and multicollinearity effects, but potentially introducing more bias (Hoerl and Kennard, 1970).

A common alternative is *Lasso regression*, or *L1 regularization*, where

$$R(\mathbf{w}) = \sum_{j=1}^{p} |w_j|.$$

Unlike ridge, this penalty function may shrink some coefficients to zero, effectively removing those predictors from the model. This makes lasso particularly appealing when we suspect that only a subset of features truly matter. In high-dimensional settings, lasso can perform both regularization and feature selection simultaneously, yielding more in-

terpretable models (Tibshirani, 1996).

Because penalties depend on coefficient magnitudes, it is standard to scale features (e.g., standardize each numeric feature to mean 0 and variance 1) before applying ridge or lasso so that the penalty treats features comparably.

Cross-validation: Choosing λ. Choosing the right value of the regularization parameter λ is crucial: when $\lambda = 0$, the model reduces to ordinary least squares and risks overfitting, while as λ grows large, the coefficients are heavily shrunk toward zero, which can lead to underfitting. A similar issue arises in choosing the type of regularization method: Ridge regression tends to perform well when many predictors contribute modestly to the outcome and are correlated with each other, whereas Lasso regression forces some coefficients to zero, effectively performing feature selection. There is no formula that tells us the optimal λ or the ideal regularization method; they must be determined empirically.

Cross-validation provides a systematic way to address both of these choices. In k-fold cross-validation, the data are divided into k subsets (folds). For a given regularization method and value of λ, the model is trained on $k-1$ folds and evaluated on the remaining fold; this process repeats k times so that each fold serves as a validation set once. The average validation error across folds gives an estimate of how well that model generalizes to new data. By repeating this process for a range of λ values, we can plot the validation error as a function of the regularization parameter. The λ that minimizes this error is then selected as the optimal penalty.

In practice, this approach not only identifies the appropriate λ but also helps determine which regularization method is best suited for the data. For instance, one might perform cross-validation for ridge and lasso (or a combination of the two, a.k.a *elastic net*) and compare their cross-validated errors. The model that yields the lowest error is then chosen.

4 Logistic Regression

Logistic regression is one of the most widely used methods for classification problems where the outcome is binary—for example, whether a borrower defaults on a loan, a customer churns, or a patient has a disease; see, for example, James et al. (2023) (Chapter 4) or Hastie, Tibshirani and Friedman (2009) (Chapter 4). The idea is to model the probability that the outcome equals one (the "positive" class) as a smooth function of the input features. Unlike linear regression, which can predict any real number, logistic regression uses the *logistic function* to ensure that predicted probabilities always lie between 0 and 1. Concretely, for a feature vector $\mathbf{x} = (x_1, \ldots, x_p)$ and weights $\mathbf{w} = (w_1, \ldots, w_p)$ with intercept w_0, the model is

$$\widehat{p}(\mathbf{x}) := \Pr(y = 1 \mid \mathbf{x}) \;=\; \sigma\left(w_0 + \sum_{j=1}^{p} w_j x_j\right), \quad \text{where } \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

is the *sigmoid* function—an S-shaped curve that smoothly maps any number to a value between 0 and 1. Very negative inputs produce probabilities near 0; very positive inputs produce probabilities near 1; and an input of zero produces 0.5. This makes it a natural way to turn a linear score into a probability.

Equivalently, logistic regression is linear in the *log-odds* (the logit):

$$\log \frac{\Pr(y = 1 \mid \mathbf{x})}{\Pr(y = 0 \mid \mathbf{x})} = w_0 + \sum_{j=1}^{p} w_j x_j.$$

Thus, increasing feature x_j by one unit multiplies the odds by e^{w_j}, holding other features fixed.

4.1 Training the model

The model learns from data by finding the set of weights that make its predicted probabilities match the observed outcomes as closely as possible. Instead of minimizing squared

errors, as in linear regression, logistic regression maximizes the likelihood that the observed data were generated by the model. This is equivalent to minimizing a function called the *logistic loss* or *cross-entropy loss*, which penalizes confident but wrong predictions much more than uncertain ones. Specifically, given training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ with $y_i \in \{0, 1\}$, the model solves

$$\min_{w_0, \mathbf{w}} \; -\sum_{i=1}^{N} \Big[y_i \log \sigma(\widehat{z}_i) \; + \; (1 - y_i) \log \big(1 - \sigma(\widehat{z}_i)\big) \Big],$$

where the linear score $\widehat{z}_i = w_0 + \sum_{j=1}^{p} w_j x_{ij}$ can be interpreted as *evidence* for class 1. Intuitively, the model adjusts its weights to push the predicted probabilities of outcomes $y = 1$ close to one, and those of outcomes $y = 0$ close to zero. While more complex than minimizing the least squares errors of a linear regression, this problem is well-behaved and can be solved using standard computational methods.

From probabilities to decisions. Once trained, the model will output the probability that a new observation belongs to the positive class using (2). For example, given the features of a loan applicant, the model will output a number between 0 and 1 representing the estimated probability of default. To make a decision, we usually apply a threshold, classifying the observation as positive if the predicted probability exceeds the threshold (see Section 1.9).

Example: Predicting customer churn. Suppose we want to build a model to predict customer churn based on some of their usage patterns—specifically, their tenure (i.e., how long they have been a customer), their average monthly bill, whether they have a 1- or 2-year contract (versus month-to-month service), whether they are enrolled in autopay, whether they have internet service, and whether they use paperless billing. Here $y = 1$ indicates that the customer churned, and the model estimates $\Pr(y = 1 \mid \mathbf{x})$ for each

customer. That is, we consider the following model;

$$\Pr(\text{churn} = 1 \mid \mathbf{x}) = \sigma\big(w_0 + w_1 \times (\text{tenure}) + w_2 \times (\text{Monthly Bill}) + w_3 \times (\text{1+yr Contract}) \\ + w_4 \times (\text{Autopay}) + w_5 \times (\text{Internet}) + w_6 \times (\text{PaperlessBilling})\big)$$

Estimating the coefficients from the churn data yields:

w_0	w_1	w_2	w_3	w_4	w_5	w_6
-1.87	-0.036	0.025	-1.23	-0.39	0.38	0.48

Thus, each additional month as a customer multiplies the odds $\Pr(\text{churn}) / \Pr(\text{no churn})$ by $e^{w_1} = e^{-0.036} \approx 0.965$ (about a 3.5% reduction in the odds per month). Having a long-term contract multiplies the odds by $e^{w_3} = e^{-1.23} \approx 0.29$ (about a 71% reduction in the odds), holding other features fixed.

Like linear regression, logistic regression assumes a linear relationship, but here it's linear in the *log-odds* of the outcome. Logistic regression works best when the log-odds of the outcome are approximately linear in the features. If the relationship is strongly nonlinear, logistic regression will struggle. Though it can still output probabilities, its accuracy will suffer unless we engineer synthetic features to capture interactions across the features and nonlinear effects. As discussed in Section 3, however, this creates the risk of overfitting, which can be mitigated using regularization techniques.

4.2 Use Cases of Logistic Regression

Logistic regression is a go-to method for binary classification in many business domains: churn prediction (predicting if a customer will leave), fraud detection (fraud vs legitimate), marketing response modeling (will a customer respond to a campaign or not), medical outcome prediction (disease vs no disease given patient stats), etc. It's favored when a quick, interpretable model is needed. For instance, a marketing analyst might use logistic regression to identify which customer attributes (age, income, engagement met-

rics) drive the probability of response to a new offer. If the decision boundary is roughly linear in the feature space (perhaps after some feature engineering), logistic regression can perform very well.

4.3 Multinomial Regression

Sometimes the outcome variable can take on more than two categories—say, predicting whether a consumer chooses product A, B, or C. The logistic regression framework can be extended to handle multiple classes. This extension is called *multinomial logistic regression*. Instead of estimating a single probability, the model estimates a probability for each class in such a way that they all add up to one; specifically, it predicts

$$\Pr(y = k \mid \mathbf{x}) \;=\; \frac{\exp\Big(w_{k0} + \sum_{j=1}^{p} w_{kj} x_j\Big)}{\sum_{\ell=1}^{K} \exp\Big(w_{\ell 0} + \sum_{j=1}^{p} w_{\ell j} x_j\Big)}, \qquad k = 1, \ldots, K.$$

Because adding the same constant to all class scores leaves the probabilities unchanged, the parameters are not identified without a normalization. A common choice is to pick a reference class (say K) and set $w_{K0} = 0$ and $w_{Kj} = 0$ for all j, so the other classes' coefficients are interpreted relative to that baseline.

The training principle is the same: it finds the set of weights that maximize the likelihood that the observed data were generated by the model. Once we have trained the model, given a set of input features, we predict that the outcome belongs to the class with the largest probability. Multinomial regression is widely used in marketing and economics to model discrete choices, as well as in machine learning tasks like text classification and image recognition.

5 k-Nearest Neighbors

The k-nearest neighbors (kNN) algorithm is a simple yet powerful supervised learning method for both classification and regression. The core intuition is that *similar examples tend to have similar outcomes*. In other words, to predict the output for a new data point, kNN looks at the k most similar data points in the training set (its "nearest neighbors") and bases its prediction on those neighbors (Cover and Hart, 1967).

This approach is often compared to how we make decisions by analogy. For example, real estate appraisers estimate a house's value by comparing it to similar recent sales ("comps"). kNN provides a clean quantitative version of this idea: given a database of past transactions, a house's value can be predicted by finding similar houses in terms of location, size, and other characteristics, and averaging their sale prices.

5.1 How kNN works

Suppose we have historical data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, where each $\mathbf{x}_i = (x_{i1}, \ldots, x_{ip})$ is a vector of p features for observation i and y_i is the outcome. In a house-price setting, $\mathbf{x}_i$ might include location, living area, lot size, number of bedrooms, sale date, etc., and y_i is the sale price. In a churn setting, $\mathbf{x}_i$ might include tenure, billing history, and contract type, and $y_i \in \{0, 1\}$ indicates whether the customer churned.

Unlike linear or logistic regression, kNN does not fit a parametric equation with estimated coefficients. For this reason it is often called an *instance-based* or *lazy* learner: the "model" is essentially the stored training data plus a definition of similarity. That said, there is still important work done *before* prediction: preprocessing the data (scaling and encoding), choosing a distance metric, and tuning hyperparameters. In practice, these choices largely determine whether kNN succeeds or fails.

Step 1: Define (and compute) similarity via a distance function. Given a new query instance with features $\mathbf{x}_q$ (and unknown outcome), kNN compares $\mathbf{x}_q$ to each training point

$\mathbf{x}_i$ using a distance function $d(\mathbf{x}_q, \mathbf{x}_i)$. The most common default is Euclidean distance:

$$d(\mathbf{x}_q, \mathbf{x}_i) \;=\; \|\mathbf{x}_q - \mathbf{x}_i\| \;=\; \sqrt{\sum_{j=1}^{p} (x_{qj} - x_{ij})^2}. \tag{3}$$

Euclidean distance is the 'straight-line' distance between two points in feature space: $\|\mathbf{x}_q - \mathbf{x}_i\| = \sqrt{\sum_{j=1}^{p}(x_{qj} - x_{ij})^2}$. Each feature contributes one squared difference; summing them and taking the square root gives the overall distance. Smaller distance means 'more similar.'

Step 2: Find the k nearest neighbors. Let $\mathcal{N}_k(\mathbf{x}_q)$ denote the set of indices of the k training points closest to $\mathbf{x}_q$ under the chosen distance:

$$\mathcal{N}_k(\mathbf{x}_q) = \left\{ i \in \{1, \ldots, N\} : \mathbf{x}_i \text{ is among the } k \text{ closest points to } \mathbf{x}_q \right\}.$$

For example, if $\mathbf{x}_1$, $\mathbf{x}_4$, and $\mathbf{x}_9$ are the 3 closest points to $\mathbf{x}_q$, then $\mathcal{N}_k(\mathbf{x}_q) = \{1, 4, 9\}$.

Step 3: Aggregate neighbors to produce a prediction. How we combine neighbor outcomes depends on whether y is continuous (regression) or categorical (classification).

Regression. The basic kNN regression prediction is the average of neighbor outcomes:

$$\widehat{y}(\mathbf{x}_q) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} y_i.$$

In other words, we take the k closest "comps" and average them.

Classification. For classification, kNN can be viewed as producing a *score* or *probability estimate* first, followed by a decision rule. For a binary label $y \in \{0, 1\}$, a natural estimate of the probability of the positive class is the fraction of neighbors labeled 1:

$$\widehat{p}(\mathbf{x}_q) = \Pr(y = 1 \mid \mathbf{x}_q) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} y_i.$$

A simple classifier is then: predict $\widehat{y} = 1$ if $\widehat{p}(\mathbf{x}_q)$ exceeds a threshold and predict $\widehat{y} = 0$ otherwise. For multiclass outcomes, we can estimate a probability for each class by taking the fraction of neighbors in that class and predict the class with the highest estimated probability.

Distance-weighted kNN The basic version above treats all k neighbors equally. A common refinement is to give closer neighbors more influence. One simple approach is to assign each neighbor i a weight

$$\omega_i(\mathbf{x}_q) = \frac{1}{\left[d(\mathbf{x}_q, \mathbf{x}_i) + \varepsilon\right]^r},$$

where $\varepsilon > 0$ prevents division by zero and $r > 0$ controls how quickly influence decays with distance. Then for a regression task, we take a weighted average of the k closest samples:

$$\widehat{y}(\mathbf{x}_q) = \frac{\sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} \omega_i(\mathbf{x}_q)}{\sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} \omega_i(\mathbf{x}_q)} \times y_i.$$

For classification, we can do a weighted vote using the same idea. Distance-weighting is especially useful when the "closest few" neighbors are much more relevant than the rest.

Scaling and encoding matter (more than for many other models). Because kNN relies directly on distances, it is extremely sensitive to the units and scales of the features. If "house size" ranges from 500 to 5,000 while "# bedrooms" ranges from 1 to 5, raw Euclidean distances will be driven almost entirely by square footage. Therefore, it is important to scale the features—typically by applying min-max normalization or standardization.

Categorical variables require encoding before we can compute distances. One-hot encoding (with 0-1 dummy variables) works well for low- and medium-cardinality categories (e.g., contract type), but for high-cardinality fields (e.g., ZIP code, product ID) it can produce very high-dimensional feature spaces. In such cases, more thoughtful fea-

ture engineering is needed; e.g., grouping rare categories, using geographic coordinates for location, using embeddings for text, and so on.

Hyperparameter tuning. Because kNN does not learn a set of coefficients, its performance depends heavily on design choices (*hyperparameters*). The main ones are the number of neighbors k, and whether we use distance-weighted aggregation.

If $k = 1$, predictions are extremely local: the model can fit noise and will often overfit. If k is very large, predictions become overly smooth (e.g., approaching the overall average for regression or the majority class for classification), which can underfit. In binary classification it is also common to choose an *odd* k to reduce tie votes. There is no universal best k, so in practice we try several values (and possibly several distance metrics) and use a validation set or cross-validation to choose the option that performs best out-of-sample.

5.2 Feature Weighting and the Distance Function

There are two different (and easy to mix up) notions of "weighting" in kNN:

- **Neighbor weighting:** closer neighbors count more in the vote/average.
- **Feature weighting:** certain dimensions count more when deciding which points are "close."

We already covered *neighbor weighting*, now we focus on *feature weighting*. In the basic version of kNN, all features are treated equally when computing distances. In many problems, however, some attributes are clearly more informative than others. For example, in real estate valuation, location and living area often matter much more than the number of bathrooms. In such cases, it can make sense to give more important features greater influence over the distance calculation.

A simple way to do this is a **weighted Euclidean distance**:

$$d_{\boldsymbol{\alpha}}(\mathbf{x}_q, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{p} \alpha_j (x_{qj} - x_{ij})^2}, \qquad \alpha_j \geq 0, \tag{4}$$

where α_j is the importance weight on feature j. Larger α_j means that differences in feature j contribute more to "how far apart" two observations are. When all weights are equal (or equal up to a constant factor), this reduces to the usual Euclidean distance and produces the same neighbor rankings.

There are several approaches to choosing feature weights. The first is to set weights based on an understanding of the setting/business; e.g., location gets high weight in real estate. The second is to treat a small number of weight settings as hyperparameters and choose what works best on the validation set or using cross-validation. Finally, there are also more advanced techniques for learning distance functions from data such as *Large Margin Nearest Neighbor (LMNN)* (Weinberger and Saul, 2009).

5.3 Strengths and Weaknesses of kNN

kNN's simplicity is one of its greatest strengths. It is easy to understand and implement, and it is easy to explain a prediction in human terms: "This house is predicted to be worth $300k because the most similar recent sales were around $300k."

Because it predicts by interpolating from nearby training examples, kNN can capture complex, nonlinear relationships without committing to an explicit functional form (unlike linear regression or logistic regression, which build in linear structure). In that sense, kNN is flexible "by default" when the data are dense enough and when the notion of similarity is well-chosen. At the same time, the very features that make kNN intuitive also create its main limitations:

1. The distance function is the entire model. kNN assumes that the features we use and the distance metric we choose truly capture what it means for two cases to be comparable. If we include many irrelevant features, we dilute similarity: points that are actually comparable may not look close in feature space. Conversely, if we omit critical features, kNN may treat very different cases as "neighbors".

2. Curse of dimensionality. As the number of features grows, distances become less informative: the nearest neighbor is often not *that* much closer than the farthest neighbor. This is why kNN typically performs best with a moderate number of features, and why feature selection or dimensionality reduction can dramatically improve performance. See Section 2.5 of Hastie, Tibshirani and Friedman (2009) for an accessible discussion of the curse of dimensionality for kNN.

3. Computational and operational costs. kNN stores the training data and does substantial computation at prediction time. If we have millions of observations, a naive implementation can be slow and memory-intensive. Indexing structures and approximate nearest-neighbor search can help, but there is still a real scalability trade-off compared to models that learn a compact set of parameters.

4. Limited extrapolation. For regression, basic kNN predictions are averages of observed outcomes in the training data. This makes kNN excellent for interpolation (i.e., staying within the range of known cases), but weaker for extrapolation (i.e., predicting what happens in truly new regimes, such as a structural break in a market).

5. Class imbalance. For classification, kNN can struggle when one class heavily outweighs others. In such cases, the majority class may dominate most neighborhoods. Common fixes include distance-weighted voting, choosing k carefully, or rebalancing the training data.

5.4 Use Cases of kNN

kNN's "look at nearby instances to decide" logic makes it a natural choice whenever similarity can be defined in a meaningful way.

Real estate and valuation ("comps"). kNN is a sensible baseline for real estate valuation: appraisers, mortgage issuers, and real estate agencies routinely benchmark a property against similar recent sales. The key is to define similarity in a way that matches market reality (e.g., location, recency, size, and relevant amenities).

Recommendation systems. A classic recommendation strategy is to find customers with similar purchase histories ("nearest neighbors" in preference space) and recommend products that those similar customers bought. Modern recommender systems often go beyond raw kNN, but neighbor-based methods remain a useful baseline and a highly interpretable way to generate recommendations.

Customer analytics and marketing response. If we want to predict whether a new customer will respond to an offer, we can look at the most similar past customers (based on demographics and behavior) and see how they responded. This provides an intuitive, example-based forecast that is easy to communicate to stakeholders.

Healthcare decision support (case-based retrieval). kNN-like approaches are sometimes used for diagnostic assistance by retrieving patients with similar clinical profiles. The model does not "discover biology", but it can highlight relevant historical analogs that support decision-making (paired with expert review).

6 Decision Trees

A decision tree is a simple yet powerful model that asks a series of questions and follows a set of *if-then* rules to form a decision or prediction. To build intuition, consider the classic game of 20 questions. One person thinks of an item, and another asks yes/no questions to figure out what it is. The goal is to choose each question to maximally reduce uncertainty about the item. Decision tree learning follows a similar strategy. For example, in the 20 questions game, a good first question might be *"Is it an animal?"* because it divides the space of possibilities roughly in half, yielding a lot of information. Starting with a very specific question like *"Is it a hippopotamus?"* would be a poor choice because a "no" answer doesn't eliminate many possibilities—at each step you want to narrow down the possibilities as much as possible. This is exactly what decision trees do: they iteratively find the most informative question to ask. Each question partitions the data into smaller subsets that are more homogeneous in terms of the target outcome than before. The process continues, asking further questions down each branch (contingent on the previous answers) until a decision can be made (Quinlan, 1986). See also Chapter 8 of James et al. (2023).

> **Decision trees in machine learning vs. decision analysis.** Many students have seen "decision trees" in decision analysis: a hand-built diagram of choices and chance events, used to compute expected values. A *machine learning decision tree* is different. It is *learned* from historical data to predict an outcome y from features $\mathbf{x}$. Its branches are chosen to improve predictive accuracy, and its "probabilities" (when it outputs them) are typically based on the fraction of training examples that fall into each leaf.

One can think of a decision tree as a flowchart that an analyst might draw to make a decision. For instance, imagine you want to predict whether a customer will **churn**. You might first ask, *"Is the customer on a month-to-month contract?"* If yes, that strongly sug-

gests a higher risk of churn (since customers with no long-term commitment might leave easily). If no (meaning the customer is on a longer contract), you would ask a different follow-up question like *"Has the customer made multiple support calls recently?"* Depending on the answer, you reach a final prediction (e.g., likely to churn or not). Likewise, if you want to estimate a house's value, a decision tree might start with the question *"Where is the house located?"* as location often has the biggest impact on real estate prices. One branch of the tree could be for houses in a city (which tend to be high-value), another for suburban/urban houses. For a suburban house, the tree might then ask *"How large is the house?"*, splitting into, say, "$> 3,000$ *sq.ft*" vs. "$< 3,000$ *sq.ft*", each leading to a predicted price range. Through this hierarchical questioning, the tree zeroes in on a value estimate (perhaps on a per square foot basis). These examples illustrate the core idea of decision trees: by asking a sequence of questions about the data's attributes, the tree gradually splits the population into smaller groups that are increasingly uniform in terms of the target outcome, until each group (leaf) is so homogeneous that a decision or value can be assigned.

Importantly, decision trees can naturally handle complex decision boundaries and combinations of conditions. For example, a churn prediction tree might discover that *either* a short contract *or* a high number of support calls can independently lead to a churn prediction. Each leaf could be seen as one scenario under which a customer churns. This flexibility means decision trees can fit a wide range of problems — in fact, any function mapping a finite set of discrete attribute values to a decision can theoretically be represented by some decision tree. At the same time, the model remains interpretable: one can follow a path and see *why* a particular decision was made (e.g., "the customer was on a month-to-month plan and had many support issues, therefore they were predicted to churn"). This interpretability and logical structure make decision trees especially appealing for applications where explaining the reasoning behind a prediction is often as important as the prediction's accuracy.

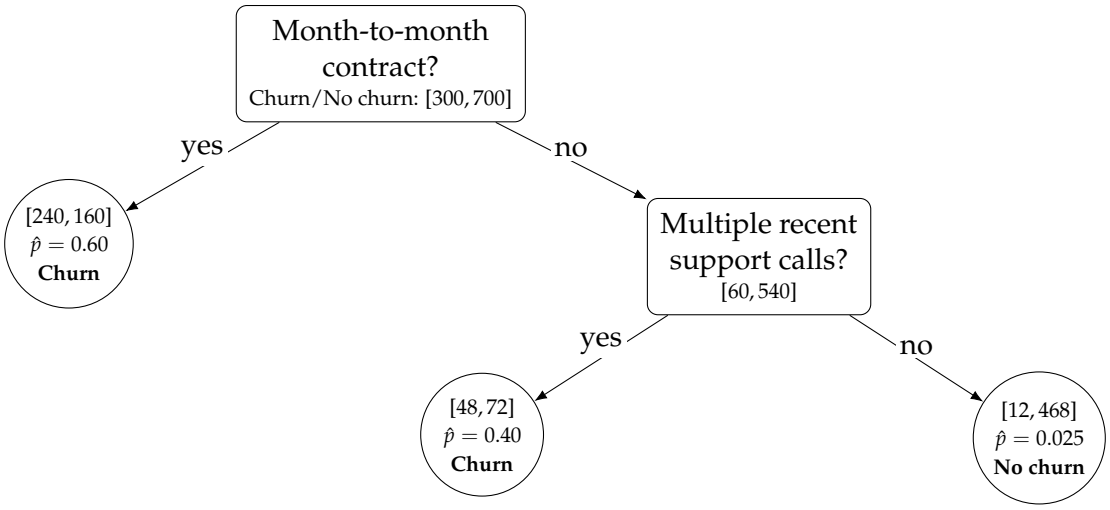


Figure 6.1: Example Decision Tree for Customer Churn (Classification). The tree is trained on historical labeled data and then used to score new customers. The root node asks whether a customer is on a month-to-month contract. Each node reports the composition of the training customers that reach that point in the tree as `[Churn, No churn]`. For example, the training set contains 300 churners and 700 non-churners. If the customer is month-to-month (left branch), the training data in that segment have `[240,160]`, corresponding to an empirical churn rate of $\hat{p} = 240/(240 + 160) = 0.60$. If the customer is *not* month-to-month (right branch), the tree asks a second question about whether the customer has made multiple recent support calls. Among those long-contract customers, the subgroup with many support calls has `[48,72]` (so $\hat{p} = 0.40$), while the subgroup with few support calls has `[12,468]` (so $\hat{p} = 0.025$). Each leaf then converts its estimated probability into a decision using a threshold: here, the model predicts *Churn* if $\hat{p} \geq 0.2$ and *No churn* otherwise.

6.1 What a tree outputs: labels, probabilities, and numeric forecasts

Using the notation from the Introduction, we observe training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, where each $\mathbf{x}_i = (x_{i1}, \ldots, x_{ip})$ is a feature vector and y_i is the outcome of interest. A decision tree partitions the feature space into *regions* (the leaves of the tree). Every observation that lands in the same leaf receives the same prediction.

Classification trees. In classification (e.g., churn vs. no churn), a leaf can output either (i) a class label or (ii) an estimated probability for each class. In binary classification with $y \in \{0, 1\}$, the most natural probability estimate is the fraction of training examples in

that leaf with $y_i = 1$:

$$\hat{p}(\mathbf{x}) = \Pr(y = 1 \mid \mathbf{x}) = \frac{1}{n_\ell} \sum_{i:\, \mathbf{x}_i \in \text{leaf } \ell} y_i,$$

where n_ℓ is the number of training observations that fall into leaf ℓ. Probabilities can then be turned into decisions, for example, using the cost-based threshold $\theta = c_{\text{FP}}/(c_{\text{FP}} + c_{\text{FN}})$.

Regression trees. In regression (e.g., predicting house prices), each leaf outputs a numeric forecast. The most common choice is the mean outcome among the training examples that land in that leaf:

$$\hat{y}(\mathbf{x}) \;=\; \frac{1}{n_\ell} \sum_{i:\, \mathbf{x}_i \in \text{leaf } \ell} y_i.$$

This makes regression-tree predictions *piecewise constant*: the tree divides the population into segments, and each segment gets its own average prediction.

6.2 Training a Decision Tree

Training a decision tree means choosing (i) *which questions to ask* and (ii) *in what order to ask them*, so that the data at the leaves becomes as **homogeneous** ("similar") as possible with respect to the outcome. As in the previous sections, we learn from labeled data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, where $\mathbf{x}_i$ is the vector of features for observation i and y_i is the corresponding outcome (a class label for classification or a number for regression).

Greedy tree growth: building top-down. Most decision-tree algorithms grow the tree *top-down*. They start at the root with *all* training observations, and at each node, they focus on the subset of observations that has reached that node (i.e., the observations that satisfy the conditions along that path). The algorithm then tries many candidate questions of the form:

$$\text{"Is feature } x_j \le \phi\text{?"} \qquad \text{or} \qquad \text{"Is feature } x_j \in A\text{?"}$$

and chooses the question that most improves prediction quality in the two child nodes.

This is a **greedy** procedure: the tree chooses the best split *right now*, at the current node, without guaranteeing that the final tree is globally optimal. Greedy methods are popular because they are fast and tend to work well in practice, but they can sometimes make an early split that looks good locally yet is not ideal in hindsight.

The key idea: "impurity" At any node, imagine you had to make a prediction *using only the observations in that node*. If the node contains almost all churners (or almost all non-churners), then it is easy to predict: we say the node has *low impurity*. If the node contains a roughly even mix of churners and non-churners, prediction is inherently harder: the node has *high impurity*.[6] One way to conceive the notion of impurity is to think in terms of the majority rule:

> *If I predicted the majority outcome for everyone in this node, what fraction of cases would I get wrong?*

If a node is 90% "no churn" and 10% "churn", then the majority rule would be wrong about 10% of the time. That is a low-impurity node. If a node is 50/50, then even the best "single-label" prediction will be wrong about half the time. That is a high-impurity node.

The splitting criterion: impurity reduction. Now suppose we are considering a candidate question at a node. That question splits the node into two child groups: a left child and a right child. A good question is one that produces child nodes that are more homogeneous than the parent. The core scoring idea is:

Impurity reduction = (impurity before split) - (weighted average impurity after split)

[6] Real implementations quantify impurity using the so-called *Gini* score or *entropy*. Gini impurity measures the probability that a randomly chosen item would be misclassified if labeled according to the node's class distribution. Entropy measures the amount of 'surprise' or uncertainty in the distribution. Both reach their minimum (zero) when a node is perfectly pure and their maximum when classes are evenly split.

Skipping the mathematical details, both of them reward splits that create child nodes that are more homogeneous than the parent.

That is, impurity reduction captures the difference between the impurity of the parent node and the size-weighted average impurity of the children nodes. Note that this weighting matters: a split that makes a tiny group perfectly homogeneous but leaves the vast majority mixed 50/50 is not very helpful.

Example (classification): Suppose at the root we have 100 customers, and churn is roughly 50/50. That is a messy starting point: no simple rule will do well. Now consider two candidate questions:

- **Split A:** "Is the customer month-to-month?"

  Imagine this produces two groups: month-to-month customers are mostly churners, while long-contract customers are mostly non-churners. Then each child group is relatively pure, so the split makes the problem much easier.

- **Split B:** "Is the monthly bill above $70?"

  Imagine this produces two groups that are still mixed 50/50. Then we have not really gained clarity; the split did not reduce impurity much.

The tree chooses Split A because it reduces impurity more: it creates segments where churn behavior is more predictable.

Group	# Churn	# No Churn	Churn rate
Month-to-month	45	15	75%
1+ year contract	5	35	12.5%

Table 3: After splitting on contract type, each subgroup is much more homogeneous than the original mixed population. The tree prefers questions that create groups like these because the outcomes become easier to predict.

Example (regression): For a regression tree, the outcome y is numeric (e.g., house price), so a node is "pure" when the prices inside it are similar. At any node, if we were forced to make *one* prediction for everyone in that node, the best single-number prediction is

the node's average price $\bar{y}_{\text{node}}$. The node's impurity is then measured by how far the individual prices are from that average, i.e., the node's *sum of squared errors*,

$$\text{SSE(node)} = \sum_{i \in \text{node}} (y_i - \bar{y}_{\text{node}})^2.$$

A split is good if it creates two child nodes whose prices are each more tightly clustered around their own averages, so the *total* SSE falls. The tree simply tries many candidate questions (e.g., "Neighborhood A?", "Living area $\leq$ 3,000 sq.ft?", etc) and chooses the one that produces the largest drop in SSE.

To make this idea concrete, if a candidate split divides a parent node into Left (L) and Right (R), the *impurity reduction* from that split is:

$$\textit{Impurity reduction} = \text{SSE(parent)} \; - \; \big[\text{SSE}(L) + \text{SSE}(R)\big].$$

For example, if the parent node contains homes from \$200K to \$2M, one average is a blunt summary, so the SSE is large. If splitting by neighborhood separates a "starter-home" area from a "luxury" area, then each child node has a much narrower price range, each child average becomes a sensible summary, and $\text{SSE}(L) + \text{SSE}(R)$ drops sharply, making the impurity reduction large. If a proposed split does *not* create tighter groups (prices remain just as scattered in both children), then the total SSE barely changes and the impurity reduction is near zero, so the tree will look for a better question.

At the end of training, each leaf predicts the average price of the training homes that landed there. The learning rule above therefore keeps splitting exactly when doing so makes those leaf averages noticeably more accurate summaries of the data.

Continuous vs. categorical features. For a continuous feature (e.g., tenure, bill amount, number of support calls), the tree tries questions like:

$$\text{“Is SupportCalls} \leq \phi\text{?”}$$

for many plausible thresholds (e.g., $\phi = 1, 2, 3, \ldots$). The best threshold is simply the one that produces the largest impurity reduction.

For a categorical feature (e.g., contract type), the tree can ask membership questions like:

$$\text{“Is ContractType Month-to-month?”}$$

Some implementations split into many branches (one per category), but many widely-used libraries use binary splits (grouping categories into two sides – yes/no) because this keeps the tree simpler and more stable.

Stopping conditions and hyperparameters. If we keep splitting until every leaf is perfectly pure, the tree can end up memorizing quirks of the training data (overfitting). In practice we control tree complexity using **hyperparameters** such as:

- maximum depth (how many questions we allow in a row),
- minimum leaf size (how many observations a leaf must contain),
- minimum improvement (only split if impurity reduction is large enough).

We typically choose these settings using a validation set or cross-validation, then report final performance on a held-out test set, consistent with the supervised learning workflow described in the Introduction.

The big picture is simple: the tree keeps asking questions as long as those questions create *meaningfully cleaner* groups, and stops when additional complexity no longer pays off out-of-sample.

6.3 Combating Overfitting

A decision tree can represent extremely complex rules. That flexibility is valuable, but it also means trees are prone to overfitting: they can learn patterns that look real in the training data but do not repeat in new data. The core symptom of overfitting is a widening gap between training performance and out-of-sample performance, and the remedy is to control complexity using validation or cross-validation.

Why trees overfit so easily. Trees split the data into smaller and smaller groups. If we keep training long enough, we can reach leaves containing only a handful of observations (or even just one). At that point, the tree is essentially *memorizing* the training data; e.g., "If customer ID 18273 and tenure 17 months and charge $49.95, then churn." Such rules can achieve very high training accuracy while generalizing poorly.

Pre-pruning (early stopping). One approach is to prevent the tree from growing too complex in the first place. Common pre-pruning controls include:

i. a maximum tree depth,

ii. a minimum number of observations required to split a node,

iii. a minimum number of observations required in each leaf, and

iv. a minimum impurity reduction required for a split.

These knobs reflect an intuitive trade-off: we only want to create a new leaf if we have enough data to support it and if it improves predictions meaningfully.

Post-pruning (grow first, then trim). An alternative is to grow a larger tree and then prune back branches that appear to fit noise. Conceptually, pruning asks: "If we replace this entire subtree with a single leaf (i.e., one simpler rule), does performance (on a held out validation set) get *substantially* worse?" If not, we prune. Many implementations use

a version of *cost-complexity pruning*, which trades off fit (training loss) against tree size; i.e. they minimize

$$\text{Training loss} \; + \; \alpha \times (\#\text{ leaves}),$$

where $\alpha \geq 0$ is a complexity penalty chosen via validation or cross-validation. The bigger the α, the more the tree will try to minimize the number of leaves.

Validation and cross-validation. Because the best depth/pruning level depends on the dataset, we treat these as hyperparameters and select them using validation (often via cross-validation), then report final performance on the test set.

A practical rule of thumb. Start with a shallow tree that produces a small number of interpretable segments. Then increase depth gradually *only if* performance is improving on the *validation set*. If performance is improving on the training set but not on the validation set, then backtrack the tree's depth—the model is likely overfitting.

6.4 Strengths and Weaknesses of Decision Trees

The biggest advantage of decision trees is *interpretability*. A tree is a set of nested *if–then* rules: you can follow a single path from the root to a leaf and explain the prediction in plain language (e.g., "month-to-month contract $\rightarrow$ many support calls $\rightarrow$ high churn risk"). In many applications, this transparency is often as important as raw accuracy because it supports communication, governance, and stakeholder buy-in.

Decision trees also require less preprocessing than many other methods. In particular, trees do not rely on distances or dot products, so feature scaling is usually unnecessary.

Another strength is that trees automatically capture *nonlinearities and interactions*. Because a tree can split on one feature and then split differently depending on that first split, it naturally expresses conditional logic ("feature A matters mainly when feature B is high"). This is exactly the kind of relationship that would require manual feature

engineering in linear models.

Trees also perform a form of *implicit feature selection*. Features that do not help reduce impurity are unlikely to appear near the top of the tree (or at all). For exploratory analysis, a small tree can be a helpful way to identify which variables appear most predictive and how they combine.

On the downside, trees have a strong tendency to *overfit* if allowed to grow deep. Deep trees can create extremely narrow segments supported by very few observations. These segments may look "perfect" in-sample but fail out-of-sample. This is why depth limits and pruning are central to using trees effectively.

Relatedly, decision trees can be *unstable*: small changes in the training data can lead to a different sequence of splits and therefore a different tree. This is a key reason why tree-based *ensembles* (like random forests and gradient boosting, covered in the following section) are so popular: by averaging many trees, ensembles produce more stable and usually more accurate predictions.

Finally, for regression problems, trees produce *piecewise-constant predictions*. This is useful when the goal is segmentation (different average outcomes for different groups), but it can be a limitation when the true relationship is smooth (as in many forecasting problems). Trees also generally do not extrapolate beyond the range of values seen in training data: they predict averages of observed outcomes, not "trend lines".

6.5 Applications: When to Use Decision Trees

Decision trees are a natural choice when the goal is to produce *actionable segments* rather than a single opaque score. Each leaf can be interpreted as a customer segment, a risk tier, or a pricing bucket, with a clear set of conditions defining membership. This makes trees appealing in contexts where managers want to ask, "Which types of customers are most at risk, and why?"

In customer churn and retention, trees are often used to identify high-risk subgroups

and the conditions associated with churn. For example, a tree might reveal that churn risk is especially high for month-to-month customers who also have recent support issues, whereas long-contract customers only become risky under different conditions. This kind of conditional story maps well to operational playbooks: different segments can receive different outreach strategies, offers, or service interventions.

In credit and risk tiering, trees provide transparent rules that can be communicated to stakeholders and (when appropriate) documented for governance. For instance, a simple tree can produce a first-pass risk tier (approve/deny, or A/B/C) using a small set of variables. A key caution, however, is that interpretability does not guarantee fairness or causality: if a tree uses a variable that is a proxy for protected attributes (e.g., race or gender), the resulting policy can be problematic. Trees make the rules visible, which is good—but you still need to audit whether those rules are appropriate.

In operations and service triage, trees work well for routing and prioritization problems: which incoming tickets should be escalated, which shipments are likely to be delayed, or which cases should be manually reviewed. In these settings, the ability to translate a prediction into a set of simple rules can reduce friction in implementation and improve adoption by frontline teams.

Decision trees are also widely used as exploratory tools. Even when a more accurate model will eventually be deployed (e.g., an ensemble), a small tree can help analysts see which variables seem predictive, identify important interactions, and sanity-check whether the model is using reasonable signals rather than leakage.

There are also settings where trees are typically a weaker choice. With *very high-dimensional sparse data* (such as bag-of-words text features), trees often struggle. For *smooth forecasting problems* where the outcome changes gradually with inputs, single trees can be too "blocky" because they predict piecewise-constant averages. And when the dataset is very small, deep trees can overfit easily unless pruning and validation are handled carefully.

7 Ensemble Methods

Ensemble methods are techniques that combine multiple models (often called learners) to solve the same problem in order to achieve better predictive performance than a single model. This is akin to the "wisdom of crowds", where many opinions are aggregated to yield a better decision. Indeed, even simple ensemble techniques can significantly improve performance. By using an ensemble, we can take relatively simple or "weak" learners and merge them together to get a more complex model that often captures the underlying patterns better.

The basic modus operandi of ensemble learning can be summarized in two steps: First, learn multiple models by training on different subsets or aspects of the training data (or in different ways), and second, combine these learned models' predictions into a final decision/prediction. For example, in a spam email filtering task, instead of trying to learn one complicated rule to identify spam, one can learn many simple rules (e.g. "the email contains the word 'free"' or "the message is very short") from different subsets of emails — each rule might be only modestly predictive on its own, but when combined they form a much more accurate spam classifier. The combination can be as simple as a majority vote or an average of the predictions.

Why do ensembles often work so well? Intuitively, different models may make different errors, so averaging their outputs can reduce variance in predictions. In fact, one major reason ensembles improve generalization is that by training models on different subsets of the data (or with different initialization), the models will see different "views" of the problem; combining them tends to average out idiosyncrasies or noise from individual models. This helps prevent any single outlier data point or noisy pattern from skewing the learned hypothesis.

Ensemble methods come in several flavors. Below is a brief taxonomy of the major types of ensemble learning techniques and their key characteristics:

- **Bagging** entails training many copies of the same base learner on different *bootstrap resamples* of the training data (i.e., datasets of the same size created by sampling *with replacement*) and then aggregating their outputs, usually by averaging or majority vote (Breiman, 1996). Bagging mainly aims to reduce variance (overfitting) by averaging out noise. A prime example is *bagged decision trees*: many trees trained on bootstrap resamples whose predictions are then aggregated (e.g., majority vote / averaged class probabilities for classification, or an average for regression). A *Random Forest* is bagged trees *plus* an additional source of diversity: at each split, the algorithm considers only a random subset of features.

- **Boosting** is a sequential ensemble approach in which we train base learners one after another, each time focusing on the training examples that previous learners misclassified. Each model is trained on a reweighted version of the data: examples that the current ensemble gets wrong are prioritized, so that the next learner is forced to concentrate on those "hard" cases. After many rounds, the weak learners are combined into a single strong predictor – typically by a weighted voting scheme where each model's vote is weighted based on its accuracy.

- One can also form **hybrid ensembles** by combining different types of models trained on the same data. For example, one might combine a decision tree, a neural network, and a logistic regression and take a majority vote. A more sophisticated version, called stacking, entails using a simple model to learn how to blend the predictions of different classifiers.

7.1 Bagging: Parallel Ensembles to Reduce Overfitting

Bagging (Bootstrap Aggregation) is one of the simplest ensemble techniques. In bagging we train several models in parallel, each on a different bootstrap resample of the training data and then aggregating their outputs, usually by averaging or majority vote. For exam-

ple, if we have N training examples, bagging might generate (say) 100 new training sets, each of size N sampled with replacement (bootstrapped) from the original data.[7] Then 100 models are trained (one per bootstrap set). For prediction, all 100 models are run on a new instance and then combined: for regression we average the numeric predictions; for classification we typically take a majority vote (or average the predicted probabilities and then pick the most likely class).

The primary benefit of bagging is that it reduces overfitting by reducing the variance of the model. Each individual model may overfit noise or particular quirks in its subset of data, but averaging many such models tends to cancel out those random errors. The intuition is that because each model is trained on a different *bootstrap resample* (with repeats and omissions), no single noisy instance will dominate *every* model's fit; when we aggregate, these idiosyncratic errors tend to wash out. In practice, bagging is especially effective for overfitting-prone learners like decision trees. A single deep decision tree can overfit, but a *Random Forest*, which is basically bagging many deep trees (with some extra randomness), yields a much more stable predictor that usually outperforms a single tree in accuracy. Ensembles of trees are so successful that random forests have become a go-to method for many tasks. They excel in many applications (e.g., finance, marketing analytics, biology, etc.) because they handle large feature sets and complex interactions automatically.

7.1.1 Training and evaluating a bagging ensemble

Bagging is straightforward to train — each base model is trained independently using the same learning algorithm on its bootstrap dataset. For evaluation, we treat the entire

[7] **Definition:** ***Bootstrapping*** is a simple way to create many different datasets from one original sample. Imagine drawing balls from an urn *with replacement*: each time you draw a ball, you record its color and then put it back before drawing again. Over many draws, some balls will be picked multiple times, while others might not be picked at all. In the same way, each "bootstrap" dataset is formed by randomly resampling the original data, so each contains repeats of some observations and omits others. This process lets us train multiple models on slightly different versions of the data, giving us a sense of how stable and reliable our predictions are.

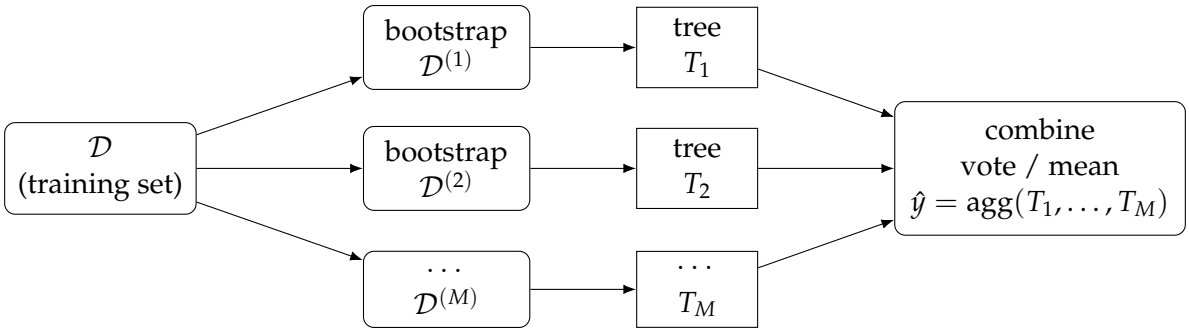


Figure 7.1: Bagging—Many models trained independently on resampled data.

ensemble as a single predictor: we can measure its performance on a validation or test set just like any other model.

Overfitting and underfitting. Bagging primarily tackles overfitting (variance). By averaging many hypotheses, it smooths out noisy fits. It is less effective at reducing bias — if the base learners are too simple to capture the true relationship, bagging many of them will still underfit. In fact, if each model has a systematic bias, the average will share that bias. Therefore, it's important to choose base models that have high variance (i.e., tend to overfit) so that bagging can do its job of variance reduction. In many cases, bagging a set of strong, deep decision trees yields an even stronger model—this is what random forests do (Breiman, 2001*a*). But bagging a set of very weak models (say, linear models for a highly nonlinear problem) won't magically make them nonlinear.

7.1.2 Strengths and weaknesses of bagging

The strengths of bagging are clear: it is simple, generally effective at improving accuracy, and robust. By averaging, it stabilizes predictions and typically reduces the risk of large errors. It's also parallelizable, making it computationally attractive for large datasets. Bagging is agnostic to the underlying learner—you can bag anything (decision trees, neural nets, even simple models)—though in practice it's most beneficial for high-variance learners. Another strength is that bagging often reduces the need for heavy regulariza-

tion because averaging stabilizes a high-variance learner. In practice, you may still apply sensible constraints (e.g., minimum leaf size for trees) to keep each base model from chasing noise.

On the flip side, a major weakness of bagging (and ensembles in general) is the loss of interpretability. Instead of one clear model, we now have, say, 100 models, so it's much harder to explain why a bagged model made a particular prediction. For example, a single decision tree can be visualized and interpreted, but a forest of 100 trees is cumbersome to interpret directly. We do have some tools (like *variable importance* measures in random forests) to get insight, but it's not as straightforward as a single model. Another weakness is that bagging is not guaranteed to help if models are too correlated: if all base learners end up learning the same patterns (e.g., if the dataset is small, bootstrap samples will overlap a lot and trees might all look similar), then averaging doesn't bring much benefit.

7.1.3 Use cases of bagging ensembles

Bagging (and random forests) works well whenever we have a sufficiently large dataset and want to improve predictive performance over a single model. It's very popular in tabular data domains; for example, predicting customer churn, credit risk modeling, medical outcomes, where decision trees perform decently. A random forest often significantly boosts accuracy and robustness in these cases, with little tuning needed. Bagging is also useful when we suspect the model is overfitting: it can stabilize high-variance models like decision trees or certain neural networks. It's an ensemble method of choice in many data mining competitions for its balance of accuracy and speed.

On the other hand, bagging may be a bad fit if model interpretability is paramount; say, if we must provide simple explanations or abide by regulatory constraints. In such cases, a single decision tree or linear model might be preferable despite lower accuracy. It's also less attractive when data is very limited: although each model is trained on (N) resampled rows, each bootstrap dataset contains fewer *unique* observations, so individual

trees can become unstable and the ensemble may not improve much. Another situation where bagging isn't helpful is if the base model is very low variance/high bias; e.g., linear regression on a truly nonlinear problem: bagging many linear regressors won't capture nonlinearity unless we add synthetic features or use a more complex model.

7.2 Boosting: Sequential Ensembles to Reduce Bias

Boosting takes a different approach from bagging: instead of training many models independently and averaging them, boosting trains models *sequentially*, with each new model specifically targeting the mistakes of the previous ones. The result is an ensemble that gradually "learns from its errors", building up a strong predictor from many simple ones.

The core idea is at each round, to identify where the current ensemble is performing poorly, and train the next model to focus on those cases. Over many rounds, each new model patches a different weakness, and the combined ensemble becomes much more accurate than any individual model. Figure 7.2 illustrates this sequential process.

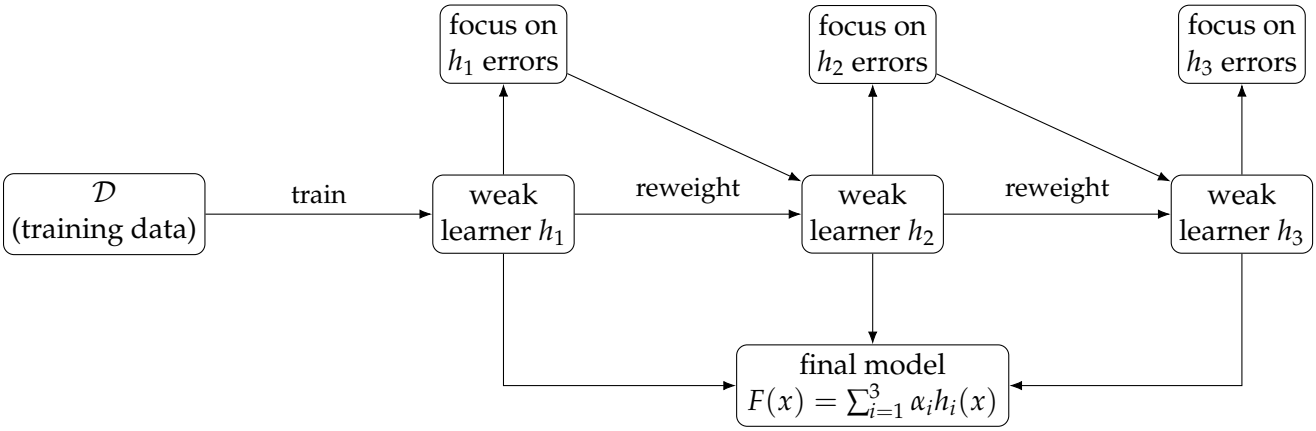


Figure 7.2: A boosting ensemble trains a sequence of simple models (weak learners) on the same dataset. After each round, the training process identifies which cases the current ensemble gets wrong and directs the next learner to focus on those "hard" cases. The final prediction combines all learners via a weighted sum, producing a strong model from many simple ones.

To build intuition, consider a churn-prediction example. In the first round, we train a very simple model—say, a decision tree with just one split—and it correctly identifies that

month-to-month customers churn at high rates but it misclassifies many long-contract customers who churn for other reasons (e.g., service complaints). In the second round, the algorithm gives extra emphasis to those misclassified long-contract churners, so the next simple tree learns a rule about complaint frequency. In the third round, the remaining errors might be customers who churn despite having long contracts *and* few complaints, perhaps because of price sensitivity—and so the next tree picks up a pricing signal. After many such rounds, the ensemble has accumulated a rich set of complementary rules, each addressing a different slice of the problem.

7.2.1 How boosting works

The base learner in boosting is deliberately kept simple—typically a *shallow decision tree* with just a few splits (sometimes called a *stump* when it has only one split). A single shallow tree is a weak learner: it captures one or two patterns and is only slightly better than random guessing. Boosting's power comes from combining hundreds of such weak learners, each one correcting for the shortcomings of the ensemble built so far. At each round, the algorithm:

1. Evaluates the current ensemble's errors on the training data.
2. Adjusts the training emphasis so that poorly predicted cases receive more attention.
3. Trains a new shallow tree on the reweighted (or residual) training data.
4. Adds the new tree to the ensemble with an appropriate weight.

The details of *how* the algorithm adjusts emphasis and assigns weights differ across boosting variants, which we discuss next, but the high-level logic—sequentially correcting errors—is shared by all of them.

7.2.2 Major boosting algorithms

Several boosting algorithms have been developed over the years. They share the sequential error-correction logic but differ in the details.

AdaBoost (Adaptive Boosting). AdaBoost is the original boosting algorithm and the simplest to understand (Freund and Schapire, 1997). It works by maintaining a set of *weights* over the training examples. Initially, all examples are weighted equally. After each round, examples that the current model misclassified receive *higher* weights, so the next model is forced to pay more attention to them. Examples that were classified correctly receive lower weights. The final prediction is a weighted vote of all the individual models, where more accurate models get a larger say.

AdaBoost is historically important and works well on clean datasets, but it has a notable weakness: because it aggressively up-weights misclassified cases, it is sensitive to label noise and outliers; i.e., if some training examples are mislabeled, AdaBoost will keep increasing their weight round after round, effectively forcing the ensemble to fit the noise.

Gradient Boosting. Gradient boosting generalizes the idea behind AdaBoost (Friedman, 2001). Instead of re-weighting examples, each new tree is trained to predict the *residual errors* of the current ensemble—that is, the gap between what the ensemble currently predicts and the true outcome. In a house-price model, if the ensemble under-predicts a home's price by $40,000, the next tree's job is to learn which features are associated with that $40,000 shortfall and correct for it. Over many rounds, the residuals get smaller and smaller, and the ensemble's predictions converge toward the true values.[8]

XGBoost. XGBoost ("Extreme Gradient Boosting") is an *engineered implementation* of gradient boosting (Chen and Guestrin, 2016). It adds several practical improvements that

[8]The name "gradient boosting" comes from the mathematical optimization technique used to derive this procedure. The technical details are not important for our purposes; what matters is the intuition of sequentially correcting errors.

make gradient boosting faster and more robust.

XGBoost was the breakout tool in applied machine learning: it dominated data-science competitions (Kaggle) for years and has been widely adopted in industry for tasks like credit scoring, fraud detection, pricing, and demand forecasting. For many structured (tabular) data problems, XGBoost remains a strong default choice.

Two more recent implementations push the engineering further:

- **LightGBM** (Ke et al., 2017) uses a different tree-growing strategy that is significantly faster than XGBoost on large datasets, often with comparable accuracy. It is a common choice when training speed or dataset size is a constraint.
- **CatBoost** (Prokhorenkova et al., 2018) adds native support for *categorical features*: instead of requiring one-hot encoding, CatBoost handles categories directly during tree construction, which can improve performance on datasets with many categorical variables (e.g., ZIP codes, product types).

Which algorithm to choose? For most applications involving tabular data, the practical differences between XGBoost, LightGBM, and CatBoost are small. A reasonable starting point is XGBoost as a general-purpose default as it is the most widely used and has the largest community of users and tutorials. LightGBM is a good choice when the dataset is very large or training speed matters, while CatBoost is commonly used when the data has many high-cardinality categorical features (e.g., hundreds of product IDs or ZIP codes). It is common practice to try two or three of these and pick whichever performs best on validation data.

7.2.3 Training and evaluating a boosting ensemble

Training a boosted ensemble is inherently sequential: each model's training depends on the results of the previous model. The main choices an analyst must make are how many

rounds to run (i.e., how many trees to add), how deep each tree should be, and how aggressively each new tree should correct the ensemble's errors.

The learning rate (shrinkage). The most important hyperparameter in boosting is the *learning rate*, also called *shrinkage*. Rather than fully trusting each new tree, the learning rate scales down its contribution—for example, a learning rate of 0.1 means each tree contributes only 10% of its full correction. This forces the ensemble to make many small, cautious adjustments rather than a few large ones, which generally leads to better generalization. The trade-off is that a smaller learning rate requires more rounds (more trees) to reach the same level of training accuracy, so training takes longer. In practice, a common starting point is a learning rate between 0.01 and 0.1, combined with several hundred to a few thousand trees, tuned via the validation set.

Early stopping. Rather than guessing the right number of rounds in advance, the standard approach is to use *early stopping*: set the maximum number of rounds high (e.g., 2,000), monitor the validation error after each round, and stop training once the validation error has not improved for a specified number of consecutive rounds (e.g., 50). The model saved at the best round is the one used for prediction. Early stopping is one of boosting's most practically useful features—it automatically determines how many rounds a given dataset needs, avoiding both underfitting (too few rounds) and overfitting (too many).

Tree depth and regularization. The trees in a boosting ensemble are typically kept shallow—a depth of 3 to 6 splits is standard. Shallow trees are weak learners by design; boosting's power comes from combining many of them, not from making any single tree complex. Deeper trees increase the ensemble's capacity but also the risk of overfitting. XGBoost, LightGBM, and CatBoost also offer additional regularization knobs (e.g., minimum number of observations per leaf, penalties on leaf weights) that further guard against overfitting. These are tuned using validation data alongside the learning rate.

Overfitting and underfitting. Boosting's main overfitting risk comes from giving the ensemble too much capacity—trees that are too deep, too many rounds, or too aggressive a learning rate—especially when the data contains label noise or outliers. Because boosting aggressively focuses on hard-to-predict cases, mislabeled examples or extreme outliers will receive increasing attention round after round, potentially causing the ensemble to fit noise rather than signal. The remedies are the ones just described: shallow trees, a moderate learning rate, early stopping, and regularization.

Underfitting is rarely a concern. If the model is not powerful enough, adding more rounds will drive training error down as long as the base learner is slightly better than random. If performance plateaus, increasing tree depth or reducing the learning rate (with more rounds) usually helps.

7.2.4 Strengths and weaknesses of boosting

Boosting algorithms (like AdaBoost, Gradient Boosting Machines, etc.) have proven to be among the most powerful out-of-the-box classifiers. The success of modern gradient boosting libraries (XGBoost and LightGBM) in Kaggle competitions and real-world problems shows that boosting is a top performer for structured data.

Boosting often achieves lower error than bagging because it directly targets mistakes and reduces bias, and generalizes well to unseen data. Moreover, it is quite flexible about the choice of base learner—we can boost decision trees, regression models, even neural networks. That said, boosting is most common and most effective with weak learners (i.e., ones that are only slightly better than random) such as shallow decision trees.

Of course, boosting isn't without its limitations: First, it can be sensitive to noise and outliers. If there are mislabeled examples or random errors, boosting will focus on them and possibly model them, harming performance, whereas bagging might dilute their influence via averaging.

A boosted model might consist of, say, a hundred trees. Thus, like bagging, the final

ensemble is complex and difficult to interpret. However, boosting can provide a ranking of which features are used most, and one can use partial dependence plots to understand which features matter most for predictions.

Lastly, boosting has several hyperparameters to tune (e.g., the number of iterations, a learning rate, the complexity of weak learners, etc), which means careful validation is necessary.

7.2.5 Use cases of boosting ensembles

Boosting is a great choice when we need a high-accuracy model and are willing to trade some interpretability and possibly training time to get it. It's been used successfully in classification and regression tasks across many areas; for example, gradient boosting machines are widely used in finance (credit scoring, forecasting), insurance, marketing, and any domain involving structured/tabular data where the relationships can be complex. These models often outperform linear models or single decision trees in terms of pure predictive accuracy. In prediction competitions or when optimizing a business metric, boosting is likely to be part of the toolkit. Boosting can also handle mixed types of data and does automatic feature selection (as weak learners will tend to ignore useless features).

On the flip side, boosting might be a poor choice if the data is extremely noisy or if each data point is very unreliable. In such cases, a regularized approach or bagging might be safer. It's also not ideal if model transparency is needed, as mentioned.

8 Support Vector Machines (SVMs)

Support Vector Machines (SVMs) are a family of supervised learning models used for classification, regression and outlier detection. They work by finding a decision boundary (for classification) or a function (for regression) that maximizes the margin between classes and relies on a few key training points called the support vectors (Cortes and Vapnik, 1995).

For illustration purposes, consider a binary classification problem for which we have historical data for two features and a corresponding label $y \in \{-1, 1\}$ for each observation.[9] In a linear SVM, the goal is to find a decision boundary (a line in two dimensions, or a hyperplane in higher dimensions) that separates the two classes *with the largest possible margin*. The margin is essentially a safety buffer: it's the distance between the decision boundary and the closest data points from each class (see Figure 8.1). SVM focuses on maximizing this buffer zone, which generally leads to a more robust classifier that can generalize better to new data.

SVMs can be applied to both classification (discrete outputs) and regression (continuous outputs) problems. The underlying principles are similar, but the interpretation of the margin and the goal of the model differ slightly between these two uses. We provide an intuitive explanation of each, highlighting their differences, their strengths, and weaknesses.

8.1 SVM for Classification

In classification, the SVM's goal is to find a hyperplane that separates the classes with the largest possible margin. In an SVM classifier, the decision boundary is the hyperplane $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = 0$, where $\mathbf{x}$ is the vector of input features, $\mathbf{w}$ is a weight vector and b is the intercept. A new point is classified by the sign of $f(\mathbf{x})$: > 0 indicates the positive

[9] Note that SVMs conventionally label classes as -1 and +1 (rather than 0 and 1) because the sign of the classifier ($f(\mathbf{x})$) directly gives the predicted class.

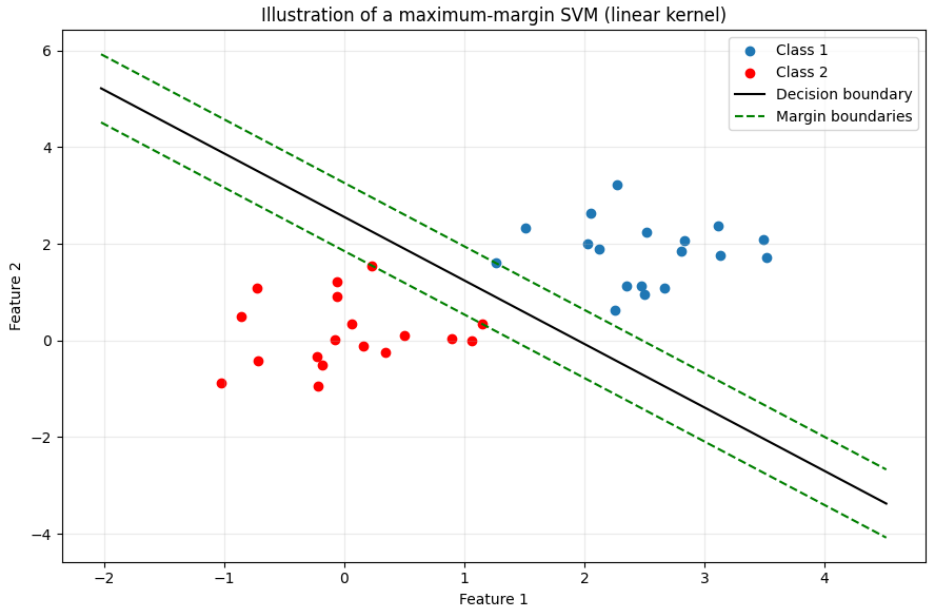


Figure 8.1: Illustration of the maximum-margin hyperplane dividing two classes. The closest points on each side are the support vectors, lying on the margin boundaries. By maximizing the margin (distance to support vectors), the SVM finds the maximum-margin decision boundary that doesn't cling too closely to any training points yet still separates the classes.

class and < 0 the negative class. The SVM training process chooses the weights and the intercept to achieve two goals: (a) ensure that all or most training points are on the correct side of the boundary, and (b) the distance from the boundary to the nearest points is maximized.

In practice, many datasets are not perfectly separable by a straight line. *Soft-margin SVM* addresses this by allowing some flexibility in classification: a few points can violate the ideal margin or even be misclassified. We introduce a parameter C to control the trade-off between maximizing the margin and minimizing classification errors. A large C value means the model penalizes misclassifications heavily – it will try to classify every training point correctly, even if that leads to a smaller margin. In contrast, a smaller C makes the model more forgiving of misclassifications, allowing a wider margin around the decision boundary. In essence, C acts like a knob for model flexibility: high C yields a tight fit to the training data (less tolerance for errors), whereas low C prioritizes a wider

margin and may ignore some outliers.

Not all classification problems can be solved with a straight-line boundary. SVMs tackle such nonlinear cases using the so-called *kernel trick*. The idea is to project or transform the original data into a higher-dimensional feature space where a linear separator does exist, but without having to create synthetic features manually. This allows the model to learn a nonlinear boundary in the original space as if it were linear in the transformed space. For example, using a polynomial kernel is like giving the model extra features that are polynomial combinations of the original features (resulting in curved decision boundaries), and an RBF (Gaussian) kernel lets the model create more complex, wiggly boundaries. The key point is that with kernels, SVMs can fit very complex relationships while still using the same efficient algorithm; all the complexity is handled inside the kernel function. Figure 8.2 illustrates the *kernel trick*, where data that is not linearly separable in its original feature space becomes separable after an implicit mapping to a higher-dimensional space.

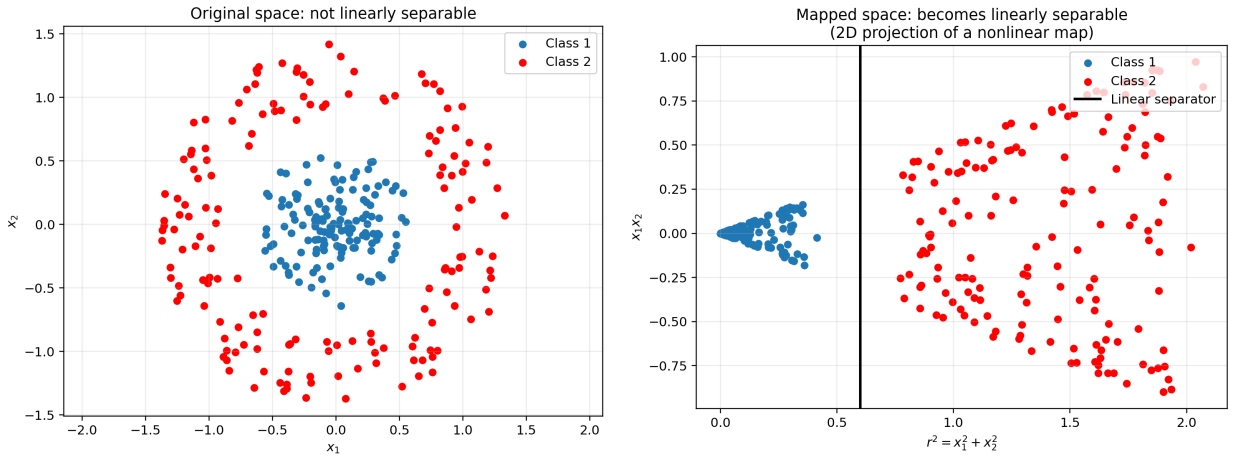


Figure 8.2: Using the kernel trick to achieve a non-linear decision boundary. **Left:** In the original 2D feature space, the classes (blue vs. red points) are arranged in a ring pattern that no single straight line can separate. **Right:** After a non-linear feature mapping, the data in the transformed feature space become linearly separable. Rather than explicitly computing this transformation for every point, SVMs use a kernel function to directly work in the high-dimensional space implicitly.

The advantage of the kernel trick is twofold: it allows us to avoid the combinatorial

explosion and overfitting risk that comes from generating a large numbers of synthetic features, and it lets the SVM focus computation on the few training cases that define the support vectors. In practice, the choice of kernel should match the kinds of patterns one expects, and the hyperparameters of the kernel can be chosen using validation.

8.2 Support Vector Regression

While SVMs are often introduced for classification, the same maximum-margin principle can be applied to regression problems through *Support Vector Regression (SVR)*. In SVR, instead of separating two classes, we try to fit a function (for example, a line) that predicts a continuous outcome. The twist is that we allow a certain tolerance, ϵ, around this function where predictions are considered "close enough" to the true values. Think of an ϵ-tube around the regression line: as long as a data point's actual value lies inside this tube (within $\pm\epsilon$ of the predicted value), we don't count that as an error. The SVR model only pays a penalty when a prediction falls outside this tube by more than ϵ. We still aim to keep the model as simple (flat) as possible, which is analogous to maximizing the margin. The parameter C again comes into play to balance complexity and errors: a higher C forces the model to fit almost all points within the ϵ-tube (lower error tolerance, potentially at the cost of a more wiggly line), whereas a lower C allows more points to lie outside the tube (higher tolerance for error) in exchange for a smoother, less complex regression function.

The result of SVR is a curve that ignores small errors and is determined by those points on or outside the ϵ tube boundaries (which play a role similar to support vectors in classification). See, for example, Figure 8.3.

SVR models can capture nonlinear relationships among the features using kernels. As before, the *kernel trick* lets SVMs exploit the benefits of synthetic features without actually having to create them manually. Common kernels are the polynomial, the Gaussian, and the RBF kernel.

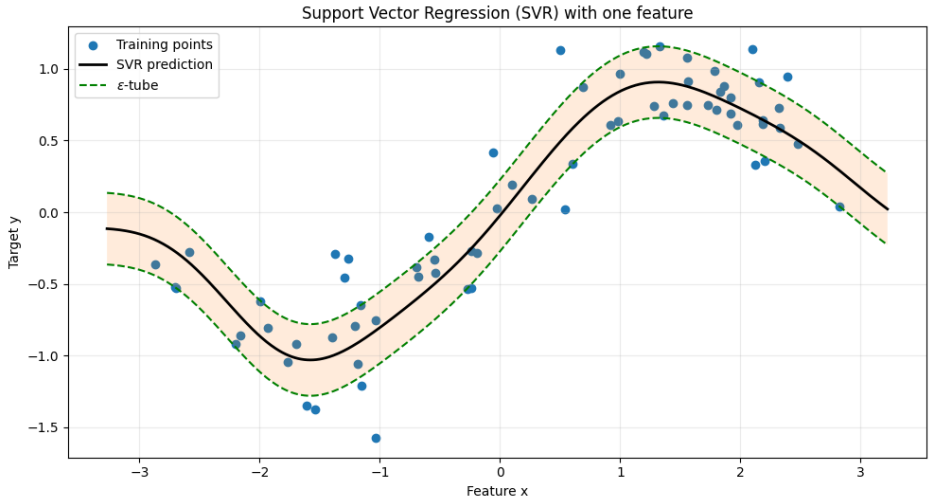


Figure 8.3: Support Vector Regression (SVR) with one feature. The solid curve is the SVR prediction $\widehat{y} = f(x)$. The dashed curves and shaded band show the ϵ-insensitive tube, $f(x) \pm \epsilon$, where deviations inside the tube are not penalized.

To make this concrete, consider the house valuation example from earlier: we want to predict house values (a continuous output) from features like size, location, and so on. We specify, say, $\epsilon = \$10,000$ as the tolerance, meaning that if the SVR predicts a price within \$10K of the actual price, we consider it "close enough" and do not count that as an error.

The SVR optimization adjusts $\mathbf{w}$ and b to both flatten the function and also pass through the data such that most houses' actual prices are within the $\pm$\$10K band of the predicted price. Houses whose prices fall outside that band (under-predicted or over-predicted by more than \$10K) will become support vectors in the regression context—these are the points that define the boundary of the tube. The objective balances making the tube as wide as possible (flat function = smaller weights = less complexity) with keeping all deviations within the tube if possible. In practice, just as with classification, often not all points can lie in the tube, so we allow some errors beyond ϵ with a penalty parameter C controlling how much penalty we assign to errors versus model simplicity.

In the house price case, extremely overpriced or underpriced houses relative to the general trend (perhaps due to unusual features or data noise) would end up as support

vectors pushing the regression line up or down, while houses that fall nicely on the general size-price trend have little effect individually.

8.3 Training, Evaluation, and Model Tuning

Training an SVM involves solving a constrained optimization problem. While mathematically and conceptually complicated, software packages to solve these problems are readily available; e.g., Python's *scikit-learn* library. The outcome of training is a model specified by $\mathbf{w}$ and b that we can then use for evaluation and prediction.

- To predict on a new instance $\mathbf{x}$, for an SVM classifier we compute $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$. If $f(\mathbf{x})$ is positive, we predict the positive class; if negative, the negative class.[10]
- For a new instance in SVR, the prediction is $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ (plus kernel terms if non-linear), which gives a real number as the estimated value.

Evaluation of SVM models is done in the same manner as other models. For classification, one would typically measure accuracy on a test set, or use metrics like precision, recall, AUC, or F1-score if the class distribution is uneven. The SVM's margin doesn't directly give a probability of class membership, but one can calibrate SVM outputs (e.g., via Platt scaling) to get probability estimates if needed. For regression, we evaluate the predictions using metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE), comparing the SVM's predicted continuous values to the actual values for a test set. SVMs do not inherently provide confidence intervals for predictions, but techniques like cross-validation can be used to estimate the likely error.

Hyperparameter tuning is an important aspect of applying SVMs, although SVMs generally have fewer hyperparameters than many other models. Key hyperparameters include the kernel choice (e.g., Linear, Polynomial of some degree, RBF, etc), and any

[10] Ties (i.e., if $f(\mathbf{x}) = 0$) can be broken arbitrarily.

kernel-specific parameters, the regularization parameter C which controls the margin-error trade-off, and for SVR, the ϵ parameter that sets the width of the no-penalty tube.

Hyperparameters are typically selected using techniques like grid search with cross-validation. For example, one might try a range of C values and kernel parameters, training the SVM on a training set and evaluating it on a validation set (or via k-fold cross-validation) to see which combination yields the best generalization performance. Fortunately, SVM performance is often not excessively sensitive to small changes in hyperparameters; e.g., there is usually a broad plateau of good C values rather than a single narrow optimum. If we choose a too complex kernel (very high-degree polynomial or very small RBF length-scale), the SVM can overfit (i.e., it will wiggle through the training points achieving close to zero training error but likely high test error). On the other hand, a too simplistic kernel might underfit. Thus, some tuning is needed, as with most machine learning models.

Feature scaling is important when training SVMs because the algorithm relies on distance measures; without scaling, variables with large numeric ranges may dominate the kernel or distance calculations, leading to suboptimal boundaries.

8.4 Strengths and Weaknesses of SVMs

SVMs are powerful, flexible models that often perform well even in high-dimensional settings (i.e., when there are many features). They are effective at finding complex decision boundaries thanks to kernel methods, yet they tend to generalize well because the margin maximization principle guards against overfitting. Another advantage is that training an SVM is a convex optimization problem, which means that if a solution exists, the algorithm will find the global optimum—there's no risk of getting stuck in a suboptimal solution as can happen with neural networks. SVMs also rely only on a subset of the training data (the support vectors) to define the model, which can make the learned model memory-efficient in deployment.

However, SVMs come with some practical challenges. They can be computationally intensive to train on very large datasets – the training time and memory usage tend to scale more poorly than for simpler algorithms as the number of data points grows. Choosing the right kernel and tuning its parameters (along with the parameter C) is critical; this hyperparameter tuning can be tricky and time-consuming, often requiring cross-validation to get right. Moreover, the results from an SVM can be harder to interpret: if a nonlinear kernel is used, the decision boundary is not easily described in terms of the original features, and SVMs don't naturally provide probability estimates for their predictions (unlike logistic regression, for example). These factors mean that while SVMs are very powerful, an analyst should weigh the added complexity against the model's performance benefits for a given task.

8.5 Use Cases of SVMs

SVMs have been successfully applied to a wide range of practical problems, both classification and regression. Financial institutions have used SVMs for credit scoring (deciding if an applicant is a good credit risk or likely to default) as well as for stock trend prediction. SVMs do well here especially when there are many indicators and possibly nonlinear interactions indicating risk. The margin maximization helps in getting a classifier that doesn't overfit to one particular historical batch of borrowers but rather finds a robust boundary. For stock or market prediction, which is more of a regression or time-series task, SVMs (and SVR) have been used to predict price movements or volatility.

SVMs have been widely used in text mining tasks such as categorizing news articles by topic or classifying emails into "spam" vs "not spam". In these problems, each document is typically represented by a very high-dimensional feature vector (for instance, a "bag of words" vector with thousands of dimensions, each indicating the presence or frequency of a particular word). The sparseness and high dimensionality of text data make SVMs a good choice, because as noted, they handle high-dimensional spaces well and the model

complexity depends on the number of support vectors, not directly on the feature count. A linear SVM often works surprisingly well for text (since a linear combination of word occurrences is often enough to separate classes like spam/ham), and training it yields a weight for each word that can be interpreted (words with large positive weight might be spam indicators, etc.). Companies like email providers and search engines historically leveraged SVMs for these tasks before the recent rise of deep learning, because SVMs were state-of-the-art in classification accuracy for many years.

In computer vision, SVMs paired with appropriate feature extraction were for a long time a leading approach. SVMs have also been used for face detection, classifying image regions as face vs non-face using kernels on image features.

In biological data analysis, SVMs are popular for tasks like classifying proteins (e.g., predict whether a protein belongs to a certain family) or diagnosing diseases from gene expression data. These datasets often have the characteristic that the number of features is huge (thousands of genes) while number of samples is limited, and there may be complex nonlinear relationships. SVMs have achieved strong results in protein function prediction and cancer classification from microarray data.

In all these use cases, some common themes emerge: SVMs are chosen when we have medium-sized datasets with potentially high-dimensional or complex feature spaces, and when accuracy is important. They have been a go-to method in competitions and benchmarks. For extremely large datasets or when interpretability is crucial, practitioners might opt for simpler or more scalable models. But SVM remains a strong baseline and often a top performer in structured data tasks.

9 Artificial Neural Networks

Deep learning is a subfield of machine learning focused on **artificial neural networks** (ANN). These networks learn complex patterns from data by adjusting internal parameters (weights) through training. The term "deep" refers to the use of many layers in the network, which enable learning of hierarchical feature representations. In essence, deep learning algorithms automatically discover the important features or factors in raw data that are needed for tasks like prediction or classification, without requiring explicit manual feature engineering. This capability has led to breakthrough results in problems previously considered very challenging for computers, such as image recognition and natural language processing. Deep learning methods have rapidly become central in artificial intelligence, powering systems ranging from voice assistants to medical image analysis. Importantly, deep learning builds on the same basic principles as simpler machine learning methods—learning from examples and generalizing, but uses large neural networks with much greater *capacity* to model complex relationships. Goodfellow, Bengio and Courville (2016) is an essential companion for neural networks material.

Neural networks are computing systems inspired loosely by the brain's networks of neurons. A neural network consists of many simple interconnected processing units called *neurons* or *nodes*. Neurons are typically arranged in layers: an input layer (receiving the input features), one or more hidden layers, and an output layer that produces the final prediction. Figure 9.1 illustrates a simple feedforward neural network—the most common type of ANNs, with three input features (x_1^0, x_2^0, x_3^0), two hidden layers with 4 neurons each, and output (y_1, y_2).

Each neuron in the first hidden layer takes the values of the input features, computes a weighted sum of these inputs, and produces an output value $(x_1^1, \ldots, x_4^1)$ after applying a nonlinear *activation* function. Moving on to the second hidden layer, each neuron takes the output values from the first hidden layer, computes a weighted sum, and applies a (possibly different) nonlinear activation function to generate another output value

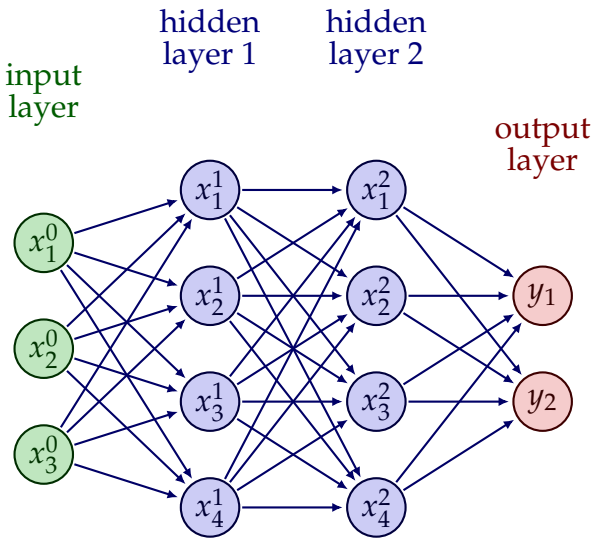


Figure 9.1: A simple feedforward neural network with three input features (x_1^0, x_2^0, x_3^0), two hidden layers with 4 neurons (nodes) each, and two outputs (y_1, y_2).

$(x_1^2, \ldots, x_4^2)$. The final prediction (y) is computed by taking a weighted sum of the output values from the second hidden layer and applying yet another activation function.

By stacking multiple layers, a network can learn increasingly abstract and complex features of the data. In fact, a famous result called the *universal approximation theorem* (essentially) shows that a neural network with sufficient structure can approximate any function with near perfect accuracy (Cybenko, 1989).

A neural network learns its behavior from training data. Initially, the connection weights that are used to compute each weighted sum (parameters) are set to random values. During training, the network processes the training data and its predicted outputs are compared to the true targets (desired outputs). The difference (error) is used to adjust the weights in a direction that reduces future errors. This learning process is typically iterative and uses an algorithm called *back-propagation* (Rumelhart, Hinton and Williams, 1986) in conjunction with an optimization method (such as stochastic gradient descent) to incrementally refine the weights. Over many examples, the network "tunes" its parameters to map inputs to correct outputs. In effect, the network automatically learns a complicated function that links inputs to outputs. Once trained, the network should be

able to generalize to new, unseen inputs and make accurate predictions. Deep neural networks often have millions of parameters and can represent extremely complex functions, which is why they perform so well on tasks with high complexity, given sufficient training data. Deep learning's power comes from this combination of large models (many layers/neurons) and learning from data, which enables computers to achieve super-human performance on some tasks.

Deep learning enjoys a huge amount of research and engineering effort worldwide. This means the tools are rapidly improving, and one can often find pre-trained models or open-source implementations to get started on a task. There are also many success stories and established best practices. This ecosystem strength means faster development and the ability to solve problems that otherwise would be very hard to tackle with in-house resources alone.

9.1 Feedforward Neural Networks

Feedforward neural networks are the foundational architecture of deep learning. In a feedforward network, information flows from the input layer through one or more hidden layers to the output layer, without cycling back. Each neuron in one layer is typically connected to *all* neurons in the next layer.

As an example, consider the feedforward neural network depicted in Figure 9.1. Each neuron in the first hidden layer computes

$$x_i^1 = g^1(w_0^{1,i} + \mathbf{w}^{1,i} \cdot \mathbf{x}^0),$$

where g^1 is an *activation* function (discussed below), and $(w_0^{1,i}, \mathbf{w}^{1,i})$ are weights.[11] Then each neuron in the second hidden layer computes $x_i^2 = g^2(w_0^{2,i} + \mathbf{w}^{2,i} \cdot \mathbf{x}^1)$, where g^2 is the activation function of the second layer and $\mathbf{x}^1 = \{x_1^1, x_2^1, x_3^1, x_4^1\}$ are the values of the

[11] Recall that $\mathbf{w}^{1,i} = (w_1^{1,i}, \ldots, w_p^{1,i})$ is a vector and we write $\mathbf{w}^{1,i} \cdot \mathbf{x}$ as shorthand for $\sum_{j=1}^{p} w_j^{1,i} x_j$.

neurons of the first hidden layer. Finally, the output is computed as $y = g^3\left(w_0^3 + \mathbf{w}^3 \cdot \mathbf{x}^2\right)$, where g^3 is yet another activation function.

Activation functions. These functions are design choices (hyperparameters), but in practice there are strong defaults. Their role is to introduce *nonlinearity* into the network, enabling it to capture complex relationships in the data. For the hidden layers (i.e., g^1 and g^2), the standard choice is the *rectified linear unit (ReLU)*, defined as $g(z) = \max\{0, z\}$, because it is simple and helps avoid common training pathologies such as vanishing gradients. For the output layer (i.e., g^3), the choice of activation function should be determined by the nature of the prediction task. Table 4 provides common default choices for a range of tasks.

Problem Type	**Default Output Activation Function**
Regression	Linear: $g(z) = z$
Binary classification	Sigmoid (interpreted as $P(Y = 1 \mid X)$)
K-class classification ($K \geq 3$)	Softmax: $p_i = e^{z_i} / \sum_{j=1}^{K} e^{z_j}$ (probability of class i)

Table 4: Common activation functions for each prediction task.

In effect, the neural network implicitly defines a function $f(\mathbf{x};\, \mathbf{w})$ that depends on the weights (here we write $\mathbf{w}$ as shorthand for all the weights). When the network receives an input, it is represented as a set of numerical features ($\mathbf{x}$), and for given weights, it generates a prediction $\widehat{y} = f(\mathbf{x};\, \mathbf{w})$. The goal of training is to find the weights that minimize a loss function. A well-trained model will have learned general patterns that apply to both the training data and new data (this property is generalization). The ability to generalize is what makes neural networks powerful—they effectively build an internal model of the data's structure that can be applied broadly, not just to the training samples. This learning process, however, requires a lot of data and computational iterations, which is why deep learning became practical only in recent years with big datasets and powerful processors.

9.2 Data Preparation for Deep Learning

The preprocessing pipeline described in Section 2—cleaning, splitting, encoding, scaling, and feature engineering—applies fully to deep learning. Two points deserve extra emphasis. First, neural networks are especially sensitive to feature scale: unscaled inputs can cause training to diverge or converge very slowly, so standardization (or min–max scaling) is virtually always necessary. Second, the validation set plays a particularly important role in deep learning because it is used to decide *when to stop training* (early stopping) and to tune architectural choices. The same leakage principles apply: fit all preprocessing on the training set and apply those transformations to validation and test data without refitting.

9.3 Training Deep Networks and Hyperparameter Tuning

Once data is prepared and a neural network architecture is chosen (i.e., the number of hidden layers, the number of neurons per layer, and the activation functions), the next step is training the model. We will explain how the training process works in simple terms, how we evaluate the model's performance, and how we go about tuning the various hyperparameters to improve results. The goal here is to demystify the training of a deep learning model without delving into heavy mathematical detail.

9.3.1 Training

In this subsection, we outline the process of training a neural network.

1. **Initialization:** We start by initializing all the weights in the network to small random numbers. At this point, the network is essentially guessing randomly when it makes predictions.

2. **Forward Pass:** We feed a *batch* of training samples (input data) into the network and produce outputs (predictions). For example, if we are training a network to predict

house values based on their features, we might feed the features of 1,000 houses whose value we know and the network produces a prediction for each house's value.

3. **Compute Loss (Error):** We compare the network's output for each sample with the true value for that sample. Using a predefined **loss function**, we quantify the error. For instance, for regression problems a common default is *mean-squared error* which, for each sample, takes the actual value y and the predicted value $\widehat{y}$, computes $(y - \widehat{y})^2$, and then takes the average over the batch of samples. For classification problems, a common default is the *cross-entropy* loss, which measures how far off the predicted probability distribution is from the empirical distribution of the training samples.

4. **Backward Pass & Weights Update:** Now the magic of training happens. The training algorithm uses calculus to determine in which direction to change each weight to make the output a bit more correct.[12]

5. **Weight Update:** Once we have the gradients, we update each weight a little bit in the direction that reduces the loss. The size of these updates is governed by a parameter called the *learning rate*. A higher learning rate means bigger steps (risking instability if too high), and a lower rate means smaller steps (risking slow convergence if too low).

6. **Iteration:** The above update process is one training iteration (for one batch of data). The training proceeds by iterating over many batches. One pass through the entire training dataset is called an *epoch*. We typically train for many epochs (tens,

[12] In mathematical terms, we use the *back-propagation algorithm*, which computes the *gradient* of the loss with respect to each weight in the network. If increasing a particular weight would increase the loss, the gradient for that weight is positive, indicating we should decrease that weight. If increasing the weight would decrease the loss (i.e., it would have made the prediction closer to correct), then the gradient is negative and we should increase that weight. Computing these gradients efficiently is what back-propagation does.

hundreds, or even more), or until the model's performance stops improving on a validation set (or reaches an acceptable level).

Over time, as the weights are updated, the network's predictions on the training data should become more and more accurate (meaning the loss decreases). We say the model is *converging* when additional training yields diminishing improvements.

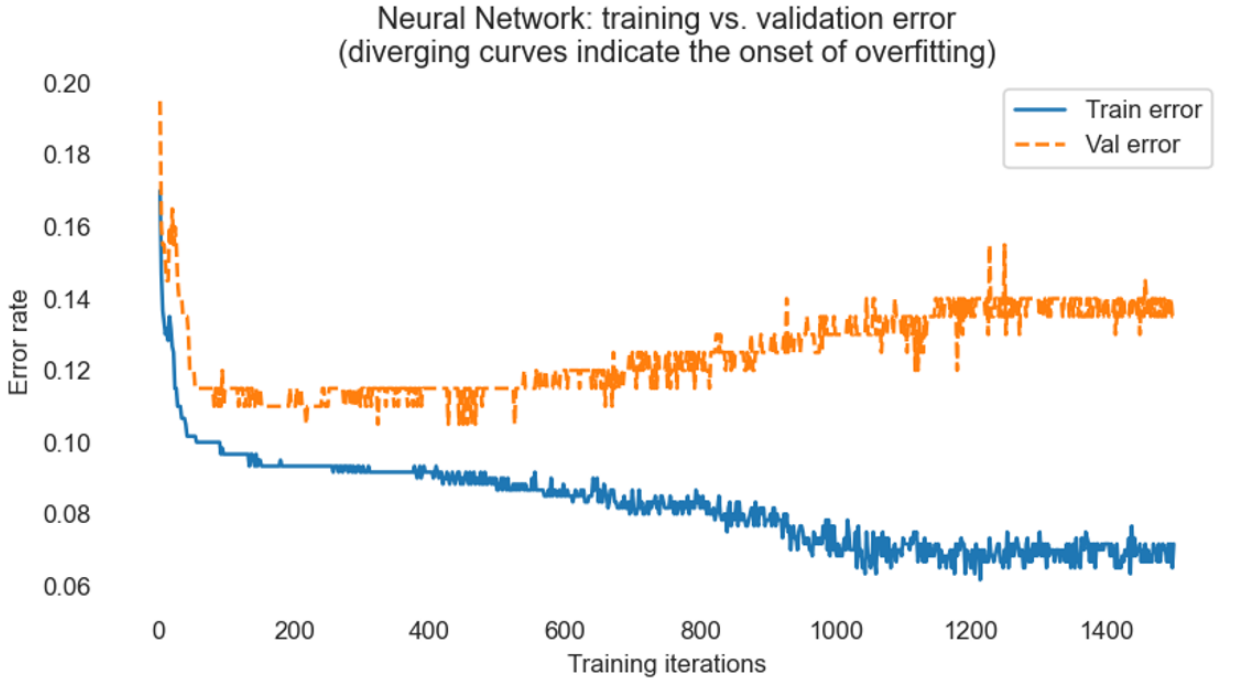


Figure 9.2: Training error versus validation error over the course of training. In the early iterations both curves fall together: the model is learning genuine patterns that generalize to unseen data. After roughly 400–500 iterations the curves diverge—the training error continues to decrease while the validation error levels off and then *rises*. This divergence is the signature of **overfitting**: the model has begun memorizing noise and idiosyncrasies in the training set rather than learning patterns that hold in new data. In practice, we would stop training near the point where the validation error is lowest and use the model saved at that point—a technique called *early stopping*.

9.3.2 Evaluation and Generalization

During training as well as after training is completed, we need to evaluate how well the neural network has learned. It is relatively easy to make a network perform well

on the data it was trained on. However, what we care about is how well it performs on new, unseen data. A model is said to *generalize* well if it has similar performance on the training and the validation/test data. A significant gap between training and test performance indicates overfitting.

Evaluation metrics. For classification tasks, accuracy (% of correct predictions) is a common metric. For skewed datasets, we might use precision, recall, F1-score (especially in problems where false negatives and false positives have different costs), or AUC. For regression problems, we might look at the mean squared error or the mean absolute error. In deep learning, often the same loss function used for training (e.g., cross-entropy loss) is monitored on the validation/test datasets, but for interpretability we convert that to accuracy or other domain-specific metrics.

Overfitting. If the model starts to overfit, the validation loss will begin to increase even as training loss continues to decline. At that point, we may apply *regularization techniques* such as *dropout*, which randomly "drops" some neurons during training to prevent reliance on any one feature, *early stopping*, which stops training after a certain number of epochs without improvement, or *weight decay* which penalizes large weights. At the end of training, the model with the best performance on the validation set is used.

When data is limited, cross-validation can be used to get a more reliable estimate of performance. However, with very large datasets common in deep learning, a single hold-out set often suffices, and cross-validation can be too time-consuming to perform with deep networks.

9.3.3 Hyperparameter Tuning

Training a deep neural network involves not only learning the weights, but also choosing a number of settings known as *hyperparameters*. Hyperparameters are not learned by the model itself, but rather must be chosen by the user. These include the learning rate,

the number of training epochs, the network architecture (i.e., the number of layers and the number of neurons per layer), the choice of activation functions, the batch size (i.e., the number of samples we feed through the network before updating weights), regularization (i.e., dropout rate, early stopping, and weight decay), and so on. Tuning these hyperparameters is partly an art and partly a science, often involving experimentation and observation. Here's how one can approach it:

- **Start with reasonable defaults:** Through practical experience in the deep learning community, there are known good starting values for many hyperparameters. For instance, one might start with a learning rate between 0.001 and 0.1, 1 or 2 hidden layers, and choose the number of neurons based on the problem complexity and data size.

- **Tweak one (or few) at a time:** A common strategy is to adjust one hyperparameter and see how it affects validation performance, then adjust another. For example, we might try learning rates 0.1, 0.01, 0.001 in separate training runs and pick the one that yields the best validation accuracy. When restricting attention to a small set of options, *grid search* (i.e., trying all combinations) can be effective. For large search spaces, more advanced methods like Bayesian optimization can be used to optimize the hyperparameters.

- **Use of the validation set:** For each candidate set of hyperparameters, we train the model, and evaluate it on the validation set. The hyperparameters with the best validation performance is chosen. It is important to not test on the true test set during tuning to avoid biasing the model. Only after finalizing the hyperparameters do we evaluate the model on the test set to get an unbiased estimate of its performance.

- **Examples of Hyperparameter Effects:**

  - Increasing the number of layers or neurons makes the model more powerful

but also more prone to overfitting if data is limited. We might gradually increase capacity until we see diminishing returns or signs of overfitting.

- A smaller batch size causes weights to be updated more frequently, making training noisier and slower but sometimes generalize better, whereas larger batches train faster but require more memory.
- If training performance is much better than validation performance, we might increase *dropout* or strengthen regularization until the gap closes.
- If training loss isn't decreasing at all, the learning rate might be too high; if it's decreasing but very slowly, we might need to increase it. Practioners often use *learning rate schedulers* that reduce the learning rate as training progresses or when validation metrics plateau, to fine-tune convergence.

Finally, one should also consider diminishing returns and deployment constraints. A slightly better set of hyperparameters might give only marginal improvement, and a more complex model might improve accuracy but at the cost of more computation and slower real-time predictions.

9.4 Specialized Neural Network Architectures

In this section, we discuss some specialized neural network architectures designed for specific types of data and problems. In particular, we will cover **recurrent neural networks**, including **transformers**, which are designed to handle sequence and time-series data, as well as **convolutional neural networks** which are especially useful for handling images, videos, and audio.

9.4.1 Neural Networks for Sequences

Many real-world data come in sequences where order matters—sentences in text, sequences of customer actions over time, daily stock prices, or weekly sales figures. Stan-

dard feedforward neural networks treat their input as a fixed-size vector with no notion of order, so they cannot naturally capture temporal patterns or long-range context. Specialized architectures have been developed to address this. We briefly discuss *Recurrent Neural Networks* (RNNs), then focus on *Transformers*, which have become the dominant architecture for sequence data.

Recurrent Neural Networks (RNNs). An RNN processes a sequence one step at a time while maintaining a *hidden state*—a vector that summarizes everything the network has seen so far. At each time step t, the RNN takes the current input $\mathbf{x}_t$ (e.g., this week's sales, price, and promotion data) together with the previous hidden state $\mathbf{h}_{t-1}$, and produces an updated hidden state $\mathbf{h}_t$. Think of the hidden state as a running summary: after processing Monday through Thursday, $\mathbf{h}_{\text{Thu}}$ encodes what the network considers important from the first four days. On Friday, the network combines Friday's new data with that summary to produce $\mathbf{h}_{\text{Fri}}$, and so on. At any point, the network can use the current hidden state to produce a prediction y_t—for example, tomorrow's expected demand.

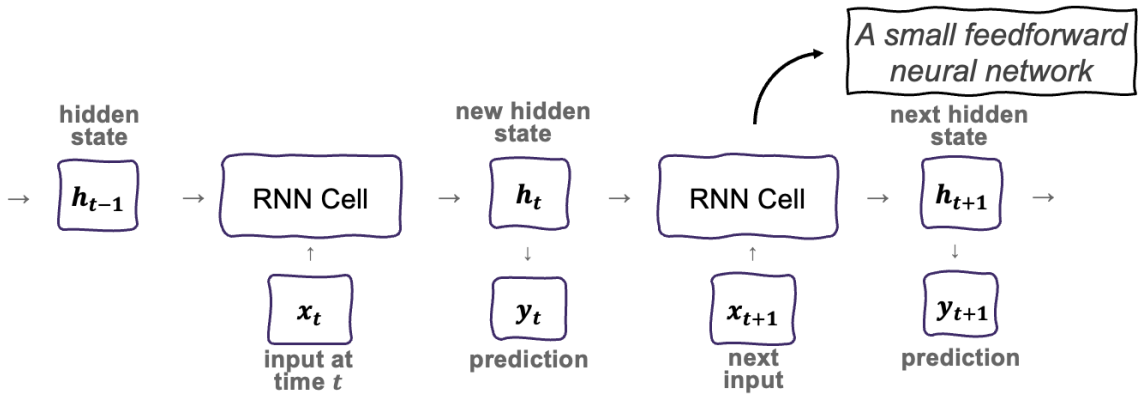


Figure 9.3: A Recurrent Neural Network (RNN) unrolled across three time steps. At each step the network reads the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$, produces an updated hidden state $\mathbf{h}_t$, and (optionally) outputs a prediction y_t. The same weights are reused at every step, allowing the network to process sequences of any length. Horizontal arrows carry the hidden state forward in time; vertical arrows show inputs and outputs.

The key strength of RNNs is their simplicity: a single small network is reused at ev-

ery time step, so the model can handle sequences of varying length. The key weakness is that older information *fades*. Because the hidden state is overwritten at every step, information from the distant past can be gradually diluted—a problem known as *vanishing memory*. Variants such as *Long Short-Term Memory (LSTM)* networks add mechanisms that help the network decide what to remember and what to forget, partially alleviating this issue (Hochreiter and Schmidhuber, 1997). LSTMs were the standard approach for sequence tasks—from demand forecasting to machine translation—from roughly 2015 to 2020. They have since been largely supplanted by transformers, which we discuss next.

Transformers. The **transformer** architecture, introduced by Vaswani et al. (2017), has become one of the most important developments in modern machine learning. It is the engine behind large language models such as ChatGPT and Claude, as well as state-of-the-art systems for translation, speech recognition, code generation, drug discovery, as well as demand forecasting, dynamic pricing, and recommendation engines. Understanding the core idea behind transformers, even at a conceptual level, is increasingly important for managers evaluating AI-powered tools and vendors.

Like an RNN, a transformer takes a sequence of inputs and produces predictions. Unlike an RNN, a transformer does *not* process the sequence one step at a time. Instead, it looks at the entire history at once and *learns which parts of the past are most relevant* to the current prediction. This ability to selectively focus on different parts of the input is called **attention**, and it is the central idea that makes transformers powerful.

Attention as a concept. To build intuition, suppose you are forecasting next week's sales for a retail product. You have the last 52 weeks of data—each week described by price, promotion status, inventory level, web traffic, and calendar flags (holiday, back-to-school, etc.). Not all 52 weeks are equally useful. A promotion two weeks ago may still be driving demand; last year's same calendar week captures seasonal patterns; a week with a stockout reveals what happens when inventory runs low. Meanwhile, a random mid-

July week with no events may be largely irrelevant.

An RNN would process these 52 weeks in order, hoping that the hidden state retains the important weeks. A transformer, by contrast, *directly computes a relevance score between the current situation and every past week*, then forms a weighted average that emphasizes the most relevant history. The weights are not hand-coded rules; they are *learned from data* during training.

How attention works. The mechanism can be understood through a simple analogy. Imagine you are in a library looking for information to write a report. You have a *question* in mind ("What happened to sales during similar promotions?"). Each book on the shelf has a *title* that summarizes what it contains, and *content* inside. To do research, you would:

1. Compare your question to the title of every book (to find relevant ones).
2. Give more weight to books whose titles closely match your question.
3. Read and combine the content from the most relevant books into a summary.

A transformer does exactly this, but in learned vector space. For each time step, it creates three internal vectors from the data:

- A **query**—what the model is looking for right now (the "question").
- A **key** for each past time step—what that past week "offers" as a match (the "title").
- A **value** for each past time step—the actual information to carry forward if that week is relevant (the "content").

The model compares the query to every key to produce a relevance score, converts the scores into weights that sum to one (so they behave like percentages), and then takes a weighted average of the value vectors. The result is a **relevance-weighted summary of the past**—a single vector that emphasizes exactly the historical information most useful for the current prediction. Figure 9.4 illustrates this process.

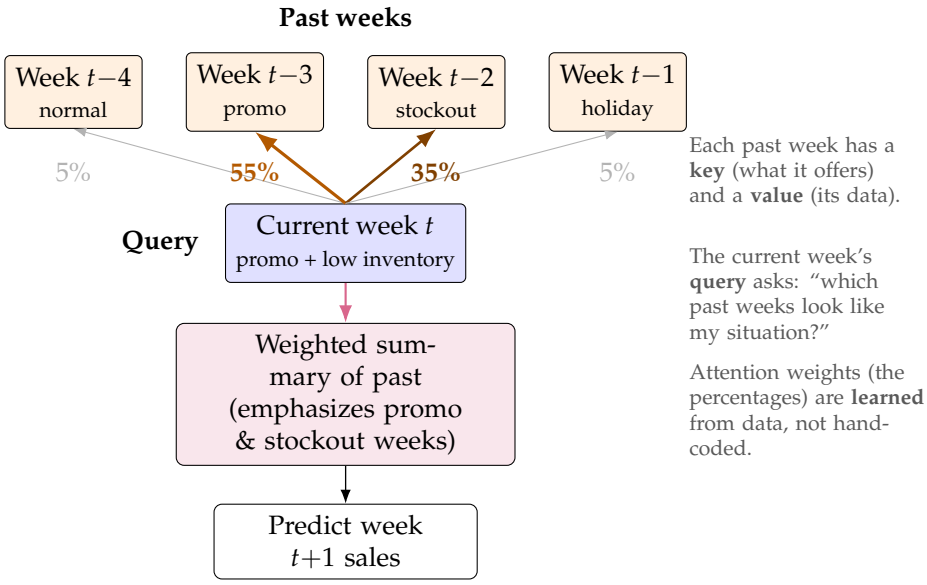


Figure 9.4: How attention works in a transformer. The current week's *query* ("promo + low inventory") is compared to the *key* of every past week. Weeks with similar conditions receive high attention weights (here, the earlier promo week gets 55% and the stockout week gets 35%), while irrelevant weeks receive little weight (5% each). The model then takes a weighted average of the past weeks' *values* to form a summary that emphasizes the most relevant history. These weights are learned from data during training—the model discovers which past patterns are informative for which current situations.

From attention to a full transformer. A complete transformer stacks several layers of this attention mechanism, interleaved with small feedforward neural networks. Each layer refines the representation further: the first layer might learn simple patterns ("last week's sales were high"), while deeper layers can capture more complex interactions ("promotions drive a two-week sales halo, but only when inventory is sufficient"). Figure 9.5 summarizes the full pipeline from input to forecast.

Transformers in practice: scale and prevalence. The transformer architecture has proven remarkably versatile and scalable. In natural language processing, it is the basis of every major large language model: ChatGPT, Claude, Gemini, and Llama are all transformers trained on massive text corpora. These models have demonstrated that scaling up transformer size and training data produces steady improvements in capability—a finding that has driven billions of dollars of investment in AI infrastructure.

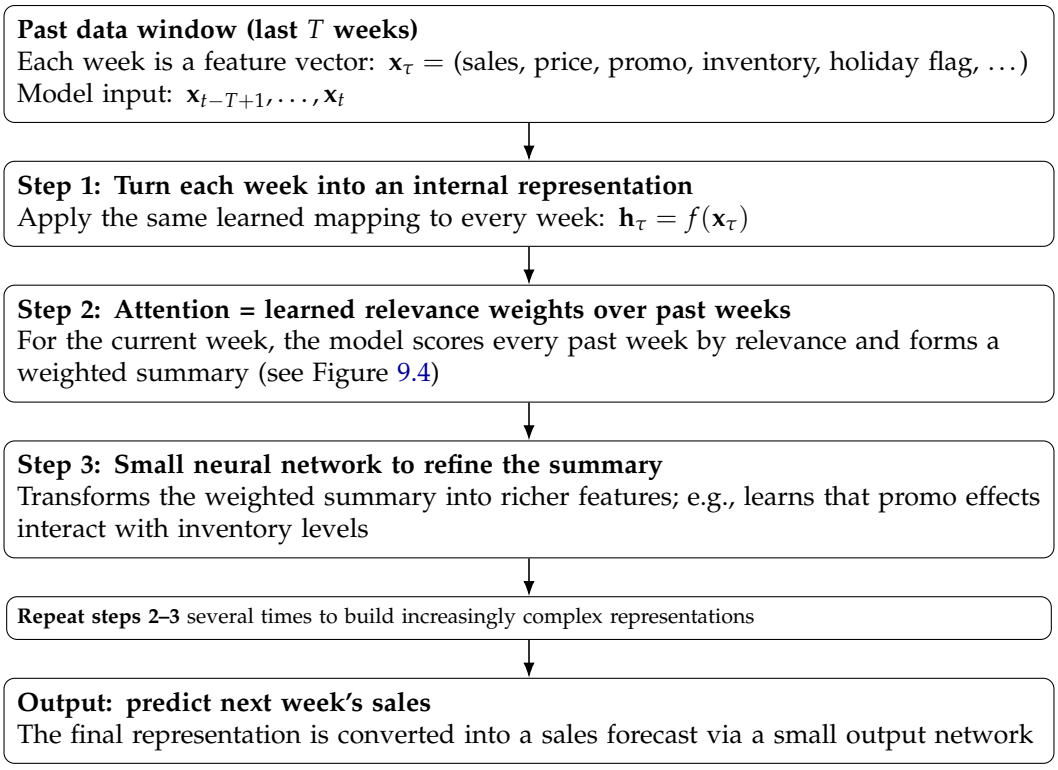


Figure 9.5: A transformer for sales forecasting. The model reads a rolling window of weekly data, learns which past weeks are most relevant to the current situation (attention), refines that information through multiple layers, and outputs a forecast.

Beyond language, transformers have extensively been used in many business applications including time-series forecasting (e.g., Amazon's Chronos and Google's TimesFM use transformers to forecast demand, revenue, and supply-chain quantities) and recommendation systems (e.g., platforms such as YouTube, Spotify, and Amazon use transformer-based models to predict which items a user is most likely to engage with, treating a user's browsing or purchase history as a "sequence" the model attends over).

9.4.2 Neural Networks for Images and Audio

Convolutional Neural Networks (CNNs) are a specialized type of neural network particularly well-suited for data that can be represented in a grid-like structure. Many forms of visual and audio information fall into this category: for instance, an image is essentially a matrix of pixel values – an image can be viewed as 3 height $\times$ width matrices, one per

color channel (red, green, blue) with each cell containing a numeric pixel intensity (often stored as 0–255, and typically normalized to a scale [0,1] before training). See, for example, Figure 9.6. A video can be seen as a sequence of image frames. An audio signal can be transformed into a spectrogram, which is a 2-dimensional matrix of time vs. frequency (effectively an "image" of sound). CNNs are designed to take advantage of the spatial and/or temporal structure in such data. They do this by learning to detect local patterns in the input: for example, edges, textures or objects in an image, or certain frequency patterns in an audio spectrogram. Because the same type of features can appear anywhere in the input (an object might be located in any part of a photo, or a particular sound could occur at any time in a recording), CNNs share the same feature detectors across different regions of the input. This weight-sharing and local connectivity make CNNs both efficient and effective for high-dimensional data like images and audio.

CNNs have driven many breakthroughs in machine learning. They are the core of most modern image recognition systems; for example, distinguishing faces in photographs or detecting tumors in medical scans. CNNs are also widely used for audio processing: by learning from spectrogram representations of sound, they achieve high performance in tasks such as speech recognition and music genre classification. Beyond vision and audio, CNNs have been used to find patterns in time-series data (like stock price movements), and in healthcare to analyze one-dimensional signals such as electrocardiogram (ECG) readings to detect anomalies.

Conceptually, a CNN's architecture is composed of two stages: (i) a feature extraction stage, and (ii) a feedforward neural network stage to make predictions. In the feature extraction stage, the network's early layers (convolutional layers typically intermixed with pooling layers) automatically learn to extract useful patterns from the input data. These layers produce feature maps, which are like new representations of the input that highlight the presence of various learned features (for example, a certain edge, shape, or texture in an image) at different locations. Next, in the feedforward stage, the final set of

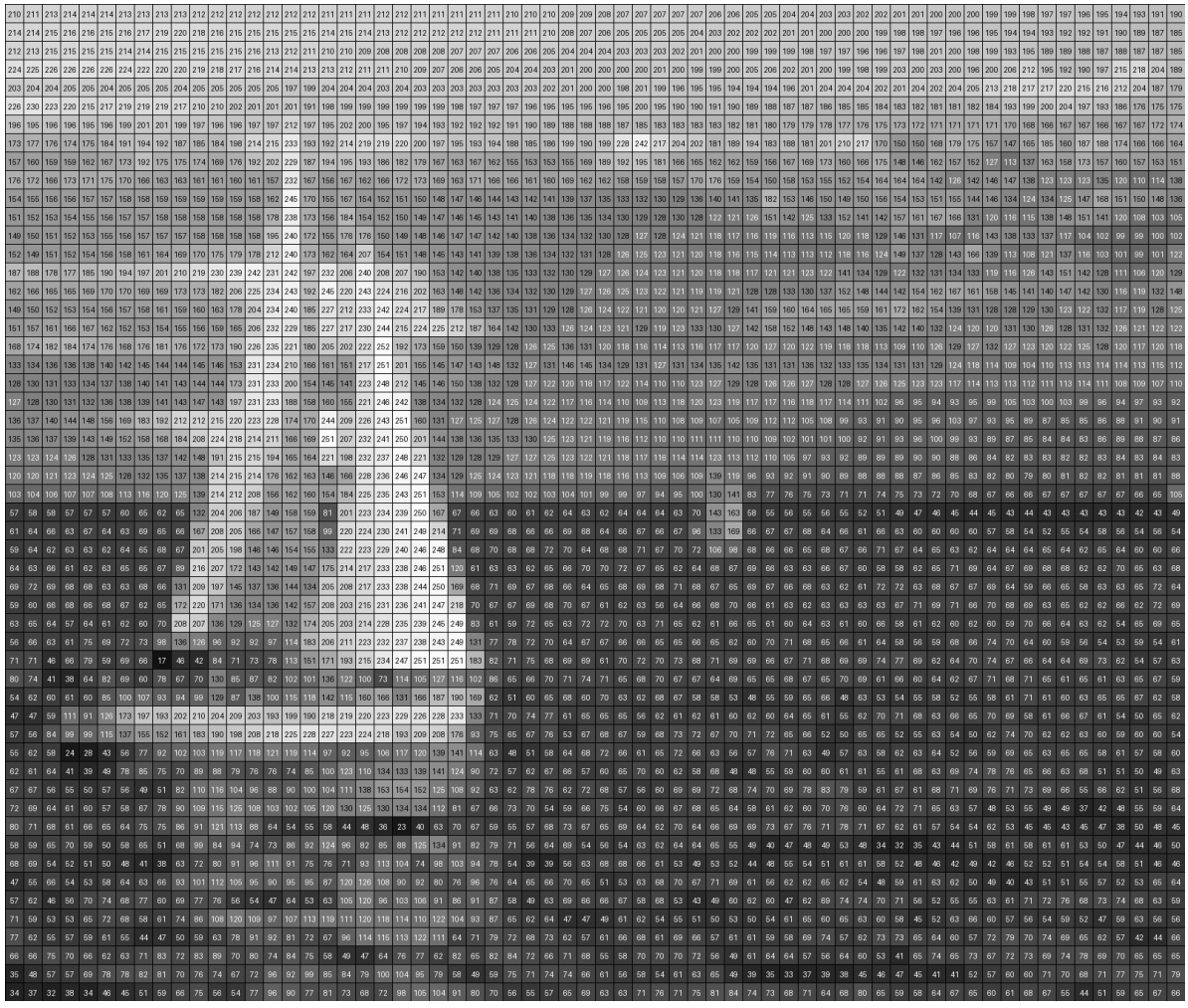


Figure 9.6: A pixel-grid representation of a sailboat at sunset: each cell is one pixel, outlined and labeled with its intensity (0–255) with brighter values tracing the sails and sky and darker values capturing the ocean.

feature maps is fed into a feedforward neural network (as covered earlier in this section). This second stage takes the distilled features and uses them to make the final prediction (such as classifying the image or signal into a category). In essence, the first part of a CNN learns what to look for in the data, and the second part decides based on those learned features.

Training. CNNs are trained end-to-end, meaning the feature extraction filters and the final classifier are learned jointly. Using labeled examples (e.g., a set of images with known labels), the training algorithm adjusts all the weights in the network, both in the convo-

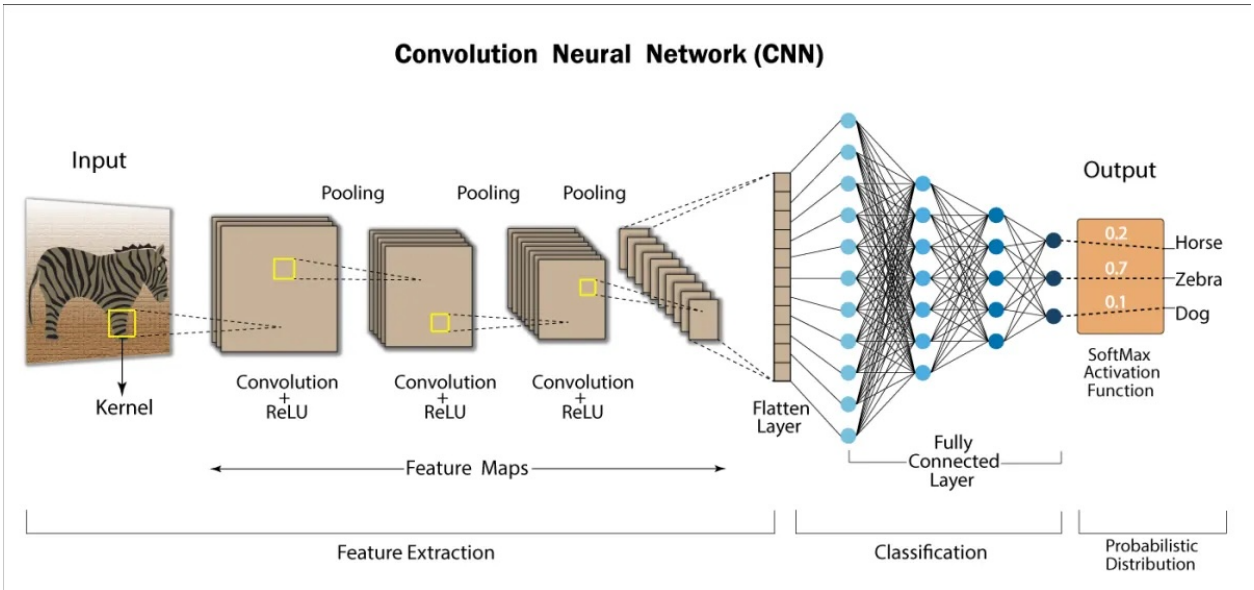


Figure 9.7: A Convolutional Neural Network. Source: developersbreach.com

lutional layers and in the fully-connected layers to minimize prediction error. Through this process, the network automatically discovers which visual or temporal features are most useful for the task, while simultaneously learning how to combine those features to make accurate predictions. CNNs also come with several hyperparameters that must be tuned. These include architectural choices such as the number of convolutional layers, the number and size of filters, the use of pooling, as well as training parameters like the learning rate and number of training epochs. Selecting good hyperparameter values (often via experimentation or validation data) is important for getting the best performance from a CNN on a given problem.

9.5 Strengths and Weaknesses of Deep Learning

Deep learning has achieved remarkable success in many fields, but it is not a silver bullet. Its value is highest when you have complex data (text, images, audio, high-dimensional signals), enough examples (or the ability to leverage pre-trained models), and a business problem where incremental accuracy creates real economic value.

A major strength is its **ability to learn complex representations**. Neural networks can

capture highly nonlinear patterns and can often learn useful features directly from raw inputs, reducing the need for manual feature engineering.

Deep learning can also deliver **high predictive performance** when paired with sufficient data and compute. In many practical benchmarks and real products (recommendations, ranking, language understanding), deep models have become the workhorse approach because they can exploit large datasets and rich signals.

Another advantage is **flexibility**: with the right architecture, neural networks can be used for regression, classification, ranking, representation learning/dimensionality reduction, and so on. For organizations, this "common toolkit" can simplify talent development and platform investment: the same training infrastructure and engineering patterns often transfer across use cases.

Finally, **prediction is usually fast after training**. The expensive part is training; once deployed, many models can score new cases quickly enough for real-time applications (subject to model size, latency constraints, and hardware).

On the downside, deep learning often suffers from **limited interpretability**. Compared to linear models or small decision trees, it can be hard to explain why a deep network made a specific decision. This is a serious issue in regulated or high-stakes contexts (e.g., credit, insurance, healthcare) where auditability, fairness, and accountability matter.

Deep learning is also **data-hungry**, especially if training from scratch. Large models with many parameters can overfit small datasets. In many business problems with only thousands (or fewer) labeled examples, simpler models may outperform deep learning in both accuracy and reliability.

A third limitation is **computational cost**. Training can require GPUs/TPUs, significant engineering time, and cloud spend. Large language models are extreme examples, but even "medium" deep models can be expensive relative to classical methods. The managerial trade-off is ROI: the accuracy gain must justify the operational complexity.

Deep networks can be **sensitive to hyperparameters and distribution shift**. They

often require careful tuning and monitoring, and performance can degrade if the data generating process changes (e.g., customer behavior, fraud patterns, product mix). This makes monitoring, retraining, and governance a core part of making deep learning work in production.

Finally, **debugging is harder than with simpler models**. When performance is poor, the cause could be data leakage, label noise, architecture mismatch, optimization instability, or a subtle preprocessing bug. Diagnosing issues is often empirical and iterative, which increases development time and creates execution risk unless teams have strong experimentation discipline.

9.6 Use Cases of Deep Learning

Deep learning has found applications in virtually every field that deals with big data. Here we highlight a range of use cases across different domains to illustrate the versatility of deep learning.

Large Language Models: Based on the transformer architecture, they are able to generate surprisingly coherent text given a prompt, which can be used for drafting emails, writing code snippets, or generating dialogue for video games. There are startups using such models to generate marketing copy or personalized messages at scale.

Time Series Forecasting and Finance: Deep learning is often applied to sequential numerical data for forecasting and anomaly detection.

- *Financial Market Prediction:* Hedge funds and trading firms experiment with deep learning (especially LSTM) models to predict stock movements or detect patterns in high-frequency trading data. (Though the unpredictable nature of markets means success is not guaranteed, it's a hot area.)

- *Credit Scoring and Fraud Detection:* Given a history of transactions, a neural network can learn to flag unusual patterns that might indicate credit card fraud or identity theft. Similarly, for loan applicants, beyond a simple credit score, a model can analyze a sequence of financial data to predict default risk.

- *Demand Forecasting:* Retailers and supply chain companies use deep learning to forecast product demand, taking into account seasonal trends, promotions, and other complex factors. RNNs or Temporal CNNs can often model these time-series better than classical linear models, especially when there are many related series (e.g., hundreds of products, where deep learning can find cross-product demand patterns).

Recommender Systems: When you see suggestions on Amazon ("Users who bought X also bought Y"), Netflix ("Top picks for you"), or Spotify's Discover Weekly playlist, deep learning is increasingly behind these recommendations. These systems take user interaction data and learn to embed users and items (products, movies, songs) into a common space where similarity means a good recommendation. For example, a model might learn that User A's taste in movies is most similar to User B and C, so it should recommend movies that B and C liked. Deep learning allows using very diverse features (text descriptions, images, user behavior sequences) to make these recommendations more accurate than older heuristic-based methods.

Computer Vision (Image & Video): One of the earliest big successes of deep learning—specifically, convolutional neural network—was in computer vision (LeCun et al., 1998 and Krizhevsky, Sutskever and Hinton, 2012). Specific use cases include:

- *Image Classification.* Identifying what objects or scenes are present in an image; e.g., classifying images of products for an online store. For instance, Facebook uses deep learning to automatically tag friends in photos by recognizing faces.

- *Object Detection.* Not just saying what is in an image, but where it is. This is used in self-driving cars to detect pedestrians, other vehicles, traffic signs, etc. It's also used in security systems to detect intruders on camera feeds or retail to count people in a store.

- *Medical Imaging.* Analyzing radiology scans (X-rays, MRIs, CT scans) to detect anomalies like tumors or fractures. Deep networks can sometimes spot subtle patterns that might be hard for the human eye, acting as a diagnostic assistant.

- *Video Analysis.* Understanding video in applications like surveillance (abnormal event detection), entertainment (automatic highlight reel generation from sports footage), or content moderation (flagging inappropriate content on social platforms).

Natural Language Processing (NLP): Deep learning has revolutionized NLP through architectures like recurrent networks and transformers.

- *Machine Translation.* Systems like Google Translate shifted from phrase-based statistical methods to deep learning, massively improving translation quality.

- *Sentiment Analysis.* Analyzing text like customer reviews, tweets, and comments to determine sentiment or emotion. Businesses use this to gauge customer satisfaction or brand perception automatically.

- *Chatbots and Virtual Assistants.* From Siri and Alexa to customer service chatbots on websites, deep learning models interpret user queries and generate relevant responses.

- *Document Processing.* Extracting information from documents like forms, contracts, or invoices. For example, a deep learning system might read a stack of invoices and automatically pull out vendor name, date, total amount, etc.

Speech and Audio Applications:

- *Speech Recognition.* Companies use deep learning models to convert spoken language to text for transcribing calls, virtual meeting transcripts, voice command interfaces, and more.
- *Voice Assistants.* Not only recognizing speech, but also generating it (text-to-speech) with human-like intonation; e.g., WaveNet. This is used in smart speakers, language learning apps (to pronounce words), accessibility tools for the visually impaired, etc.
- *Audio Detection:* Identifying sounds; e.g., detecting gunshots or glass breaking in security systems, classifying music genres for recommendations, and in healthcare, listening to coughs to predict illnesses.

Generative Modeling and Creativity: Deep learning models can create new content.

- *Images.* GANs (Generative Adversarial Networks) can generate photorealistic images of faces, objects, even imaginary creatures. These have been used to create art, to imagine how a design (e.g., a car or fashion item) might look, or even to generate synthetic training data.
- *Music.* There are deep learning models that compose music in different styles, or that can continue a melody you provide.
- *Deepfakes:* Controversially, deep learning has been used to generate fake audio or video; e.g., making someone appear to say something they never did. This exemplifies that any powerful technology can be double-edged.

Healthcare and Life Sciences: Deep learning is also being applied to things like:

- *Drug Discovery.* Predicting how different molecules will behave or how they might bind to targets (protein structures) to assist in finding new pharmaceutical compounds faster.

- *Genomics.* Deep models are used to interpret DNA and protein sequences; e.g., by predicting which gene mutations are likely pathogenic, or how genes are regulated in a cell.

- *Personalized Medicine.* Using patient data to predict risks or recommend treatments. For example, predicting which patients are at high risk of hospital readmission and why, enabling preventative care.

Manufacturing & Agriculture: In factories, deep learning can analyze sensor data from machines to predict maintenance needs. Visually, it can inspect products on an assembly line for defects at high speed (using cameras and CNNs). In agriculture, deep learning helps in tasks like identifying plant diseases from leaf images, or in precision farming (e.g., analyzing drone footage to decide where to water or fertilize).

In summary, deep learning's ability to handle unstructured data (images, text, sound) and find patterns at scale has enabled automation of tasks that were previously thought to require human perception and intuition. It's also important to note that successful application of deep learning requires not just the algorithm, but also the infrastructure and strategy around data collection, model maintenance, and integration with business processes.

10 Interpretation and Model Selection

10.1 Interpreting Machine Learning Models

Modern machine-learning models like ensembles and neural networks often behave like black boxes. Decision-makers need to understand what drives predictions and which levers they might pull to change an outcome. There are two axes to keep in mind:

- **Global methods** summarize how a model behaves across the entire dataset—which features matter most and how do they generally influence predictions?
- **Local methods** explain a single prediction—why did the model flag *this* customer as high risk?

We focus on *model-agnostic* techniques, which treat the model as a black box and work with any algorithm, as opposed to *model-specific* techniques that take advantage of a particular model's structure.

SHapley Additive exPlanations (SHAP). SHAP values are the most widely used tool for interpreting predictive models. They are grounded in *Shapley values* from cooperative game theory (associated with Nobel Laureate Lloyd Shapley; Shapley, 1953) and answer a simple question: *for a given prediction, how much did each feature contribute—positively or negatively—relative to a baseline?*

The baseline is typically the model's average prediction across the training set. For a given observation, each feature receives a SHAP value: positive values mean that feature pushed the prediction *above* the baseline, negative values mean it pushed the prediction *below* the baseline, and values near zero mean the feature had little effect. The SHAP values for all features always sum to the difference between the observation's prediction and the baseline, which ensures the contributions "add up."

SHAP is powerful because it provides both local and global explanations using the same underlying framework. Two visualizations are especially common:

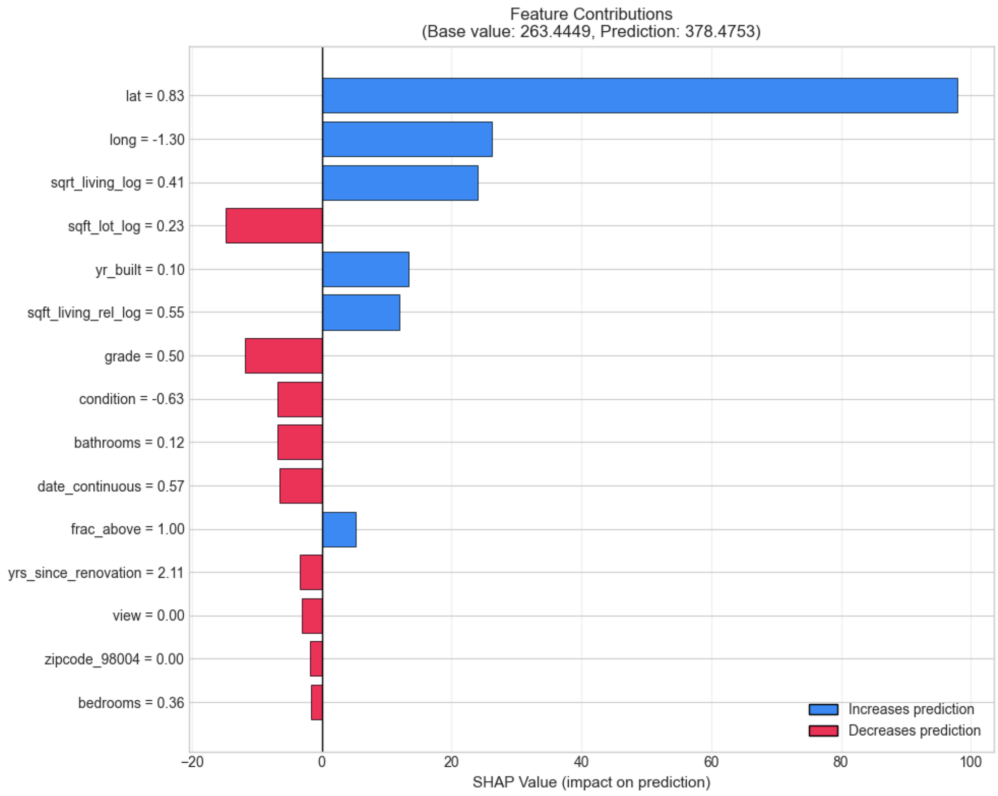


Figure 10.1: SHAP waterfall plot for a single house in a house price (per sq.ft) prediction model. The plot starts at the baseline (the average prediction) and shows how each feature pushes the prediction up (red) or down (blue), arriving at the final predicted price (per sq.ft) for this house.

- **SHAP waterfall plot** (local explanation). For a single observation, the waterfall plot starts at the baseline prediction and shows how each feature pushes the prediction up or down, step by step, until reaching the model's final output. For example, Figure 10.1 illustrates a waterfall plot for our house-price example.[13] The baseline is the average predicted price per square foot across all homes ($263). For this particular house, the latitude (a proxy for neighborhood desirability) pushes the prediction up substantially, while the relatively small living-space square footage relative to neighbors pulls it down. The individual contributions sum to the final prediction of

[13]The feature names in the plot reflect the variable names used in the analysis code. Key mappings: `sqft_living_log` is the log of living-space square footage; `sqft_living_rel_log` is the log ratio of the home's living space to its neighbors' (a relative-size measure); `lat` and `long` are latitude and longitude (proxies for neighborhood); `frac_above` is the fraction of living space above ground level; `date_continuous` is the sale date expressed as a continuous year fraction.

$378 per square foot.

- **SHAP summary plot** (global explanation). The summary plot displays *every observation's* SHAP value for *every feature* on a single chart, typically as a scatter where the x-axis is the SHAP value (how much the feature affected the prediction) and the color indicates whether the feature's actual value was high or low. This reveals both *which features matter most* (features with wide spreads of SHAP values) and *how they matter*. For example, in Figure 10.2, latitude and log living-space square footage have the widest spread, confirming that location and size are the dominant price drivers. For grade, the color pattern shows that high grade values (red dots) consistently push the prediction up, while low grade values (blue dots) push it down—exactly what we would expect.

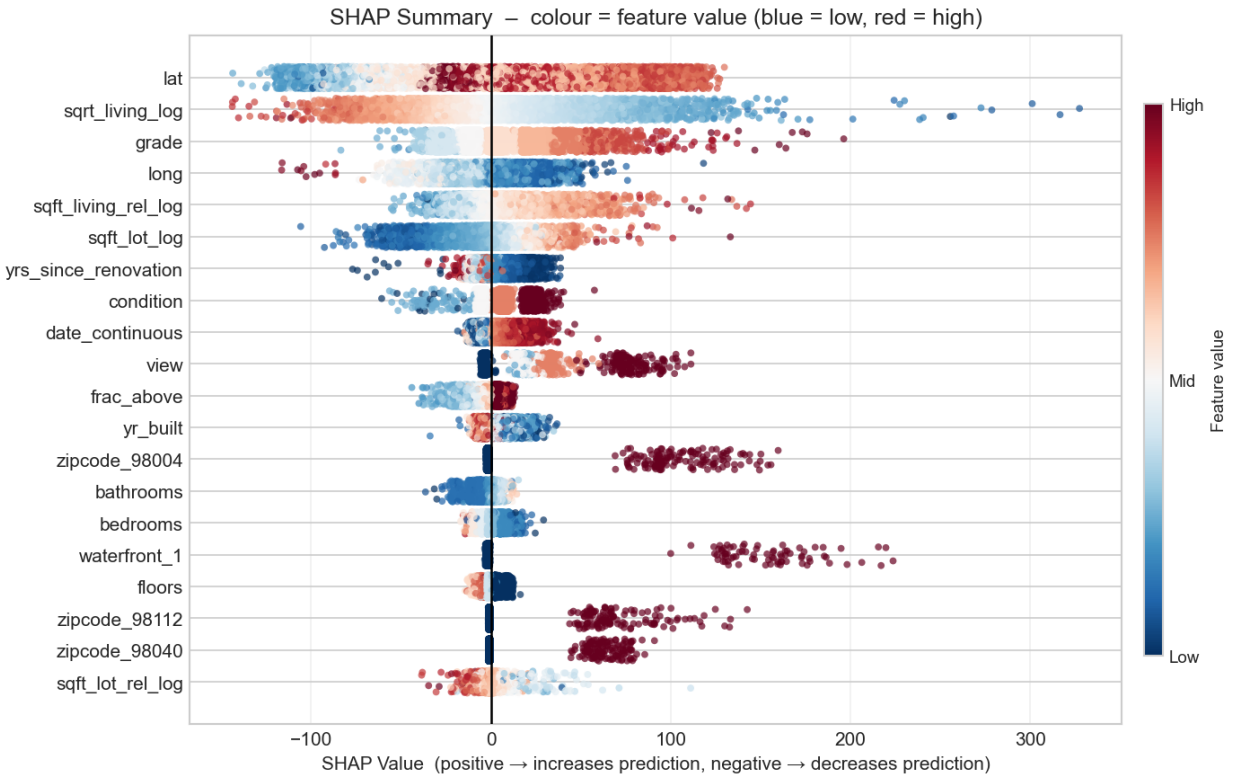


Figure 10.2: SHAP summary plot for a house price (per square foot) prediction model. Each dot is one observation; the x-axis shows the SHAP value (impact on prediction), and color indicates whether the feature value was high (red) or low (blue). Features are ranked by overall importance (top = most important).

SHAP's main limitation is that it relies on approximating what the model would predict under different "coalitions" of features. When features are correlated, some of these coalitions may represent unrealistic combinations of values (e.g., a 1,500 sq.ft house with 5 bedrooms), which can distort the explanations. Despite this caveat, SHAP remains a standard tool for model interpretation in many applied settings.

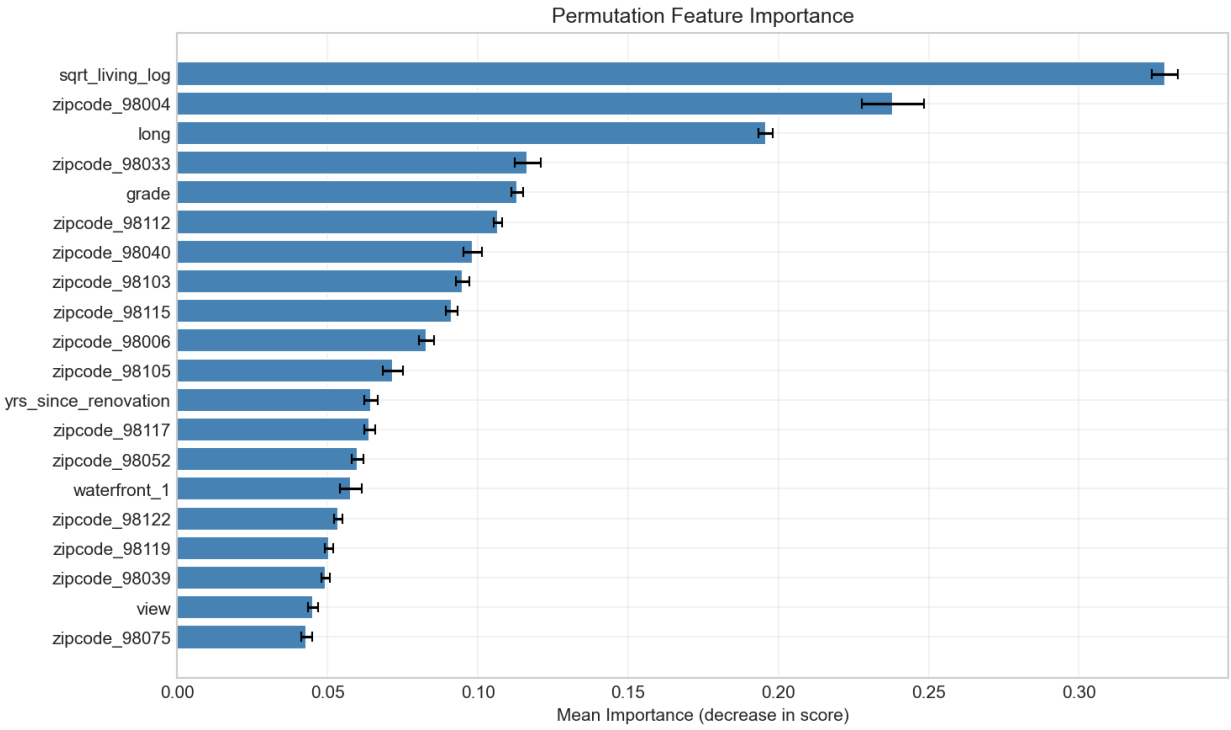


Figure 10.3: Permutation feature importance for a house price (per square foot) prediction model. Each bar shows the increase in validation error when that feature's values are randomly shuffled. Longer bars indicate features the model relies on more heavily.

Permutation feature importance (PFI). Permutation feature importance measures how much the model depends on a particular variable (Fisher, Rudin and Dominici, 2019). To compute it, you first evaluate the model's prediction error on a validation set. Then, for one feature at a time, you *shuffle* (randomly permute) the values of that column across all records—imagine randomly assigning each house's size or zip code to another house. You then feed this perturbed dataset to the model and record how much the error increases. If the error rises substantially, the model was relying on that feature; if the error barely

changes, the feature was not contributing much. Repeating the shuffle multiple times and averaging the error change provides a more stable estimate.

This technique works with any model, does not require retraining, and automatically incorporates interactions between variables (because shuffling a feature disrupts any interaction it participates in). However, PFI only tells you *how much* a feature matters, not *how* it affects the prediction; i.e., whether higher values increase or decrease the output. It can also be misleading when features are highly correlated: shuffling one feature in a correlated pair can understate its importance because the other feature tends to co-move in the data.

Partial dependence plots (PDP). A partial dependence plot summarizes how changing one (or two) features affects the model's prediction *on average*, holding everything else constant (Friedman, 2001). To construct a PDP, you choose a feature and a grid of values to explore. For each grid value, you create a hypothetical dataset by replacing that feature's value in every observation with the chosen value, while leaving all other features at their original observed values. You then have the model make predictions on this modified dataset and average the predictions across all records. Repeating this for each grid value yields a curve showing how the average prediction changes as the feature varies.

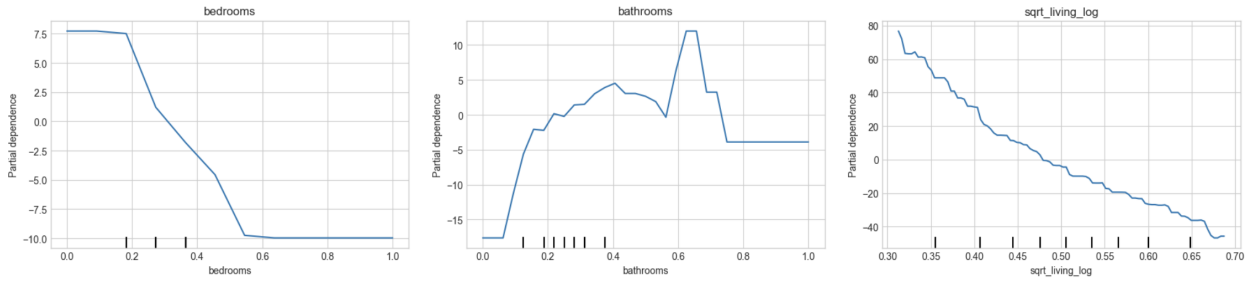


Figure 10.4: Partial dependence plots showing how the predicted house price per sq.ft changes as three features vary, averaging over all other features. *Left:* the predicted price per sq.ft declines as the number of bedrooms increases—likely because, holding total square footage fixed, more bedrooms means each room is smaller (see the multicollinearity discussion in Section 3.2). *Right:* the predicted price rises steeply with log living-space square footage, flattening at the high end, consistent with the diminishing-returns pattern that motivated the log transformation in Section 2.8.

PDPs are easy to interpret and reveal whether the model's learned relationship with a variable is linear, monotonic, or more complex. However, they have two important limitations. First, PDPs assume that the feature being varied is independent of all others; when features are correlated, the plot averages over unrealistic combinations, which can bias the curve. Second, the method shows only the *average* effect: heterogeneous responses among different subpopulations can cancel out, producing a flat curve even when the feature matters in opposite directions for different groups.

Individual conditional expectation (ICE) plots. ICE plots refine PDPs by displaying *one line per observation*, showing how that individual's predicted outcome changes as a feature varies (Goldstein et al., 2015). For each instance, you hold all other features fixed at their actual values, replace the feature of interest with values from a grid, and record the model's predictions. Plotting these lines together reveals heterogeneity: some customers might show a steep increase in predicted churn risk when monthly charges rise, while others barely change. When the ICE lines are roughly parallel, the PDP (which averages them) is a good summary. When the lines cross or fan out, the feature's effect depends on other characteristics—an interaction the PDP alone would hide.

ICE plots are intuitive and can uncover interactions, but they share PDP's limitation when features are correlated: holding one feature at its actual value while varying another can create implausible combinations.

Counterfactual explanations and recourse. Counterfactual explanations ask: "What is the *smallest change* to this individual's characteristics that would lead the model to make a different decision?" (Wachter, Mittelstadt and Russell, 2018) For example, if a loan application was rejected, a counterfactual might say that raising annual income by $10,000 while reducing the number of existing credit lines by two would result in approval. In practice, you define a desired outcome (e.g., approval, or a probability threshold) and then search for a modified version of the instance that produces this outcome while being

as close as possible to the original across all features.

Counterfactuals are attractive because they are actionable and intuitive: they tell decision-subjects exactly which levers to pull, and they can be generated without access to the underlying training data or model internals—only the prediction function is needed. They are widely used in credit and hiring decisions, where applicants may have a right to know what would change the outcome. However, counterfactuals come with challenges. For a single instance there are often many possible counterfactuals, and choosing which to present requires judgment. They may also suggest changes that are unrealistic (e.g., changing someone's age) unless the search is constrained to feasible and actionable features.

Choosing and using interpretability tools. These methods are complementary rather than mutually exclusive. A useful practice is to follow a layered approach:

1. **Start with global explanations.** Use permutation feature importance to identify which features are most influential for predictive performance. Use SHAP summary plots to obtain a global view of feature importance together with the direction and distribution of feature effects across observations. Use partial dependence plots (PDPs) to visualize how features are associated with predictions *on average*. If predictors are strongly correlated, interpret these tools cautiously: permutation importance may understate the importance of correlated variables, and PDPs may average over unrealistic feature combinations.

2. **Then examine local behavior and heterogeneity.** Use SHAP waterfall plots to decompose an individual prediction into feature-level contributions relative to a baseline. Use individual conditional expectation (ICE) curves to see how the prediction for a particular observation changes as one feature varies, holding the remaining features fixed. ICE curves are especially useful for detecting heterogeneity and interactions that may be hidden by average PDP effects, although they are visually

heavier and can be harder to read when many observations are plotted together.

3. **Finally, provide recourse when relevant.** If stakeholders need to know what changes would lead to a different prediction, use counterfactual explanations. Such explanations should be restricted to features that are actionable and feasible to change.

Throughout, it is important to remember that these tools explain *model behavior*, not necessarily real-world causality. These tools describe patterns the model has learned from the data; they don't establish that changing a feature would *cause* the outcome to change in the real world. Answering causal questions requires a separate research design, such as an experiment or a credible quasi-experimental strategy, in addition to a predictive model. For a comprehensive discussion of interpretability tools for machine learning, see Molnar (2022).

10.2 Guidelines for Model Selection

Selecting a supervised learning model begins by examining the data and the business problem rather than picking a favorite algorithm.

Dataset size. The size of the training set matters a lot. With millions of examples, scalable methods like linear models, gradient-boosted trees, or neural networks handle the volume well, whereas methods like k-nearest neighbors or SVMs with nonlinear kernels can become prohibitively slow. In contrast, if you have only a few hundred or a few thousand data points, flexible models such as large neural networks will overfit; simpler algorithms or ones with strong regularization should be favored.

Data quality and complexity. Noisy data—records with measurement errors, mislabeled examples, or heavy outlier contamination—calls for models that regularize aggressively and avoid fitting the noise. Regularized linear models, soft-margin SVMs, and bagged trees are robust choices. If you suspect nonlinear relationships or interactions

among features, it makes sense to move beyond linear models: tree-based ensembles, neural networks, or SVMs with nonlinear kernels.

Interpretability requirements. How interpretable the model must be depends on the industry and stakeholders. In finance, healthcare, and policy settings, regulators and end-users may require explanations for individual decisions. Here, linear/logistic regression and decision trees are natural choices, although more complex models can be interpreted post hoc using the tools discussed in Section 10.1. In contexts where interpretability is less critical—such as product recommendations or ad ranking—ensembles and deep neural networks tend to be preferred for their higher accuracy.

Practical constraints. Training time, computational resources, and the deployment environment also influence the choice. Simpler algorithms train faster and are easier to iterate on, making them ideal for rapid prototyping. If the model must run on a mobile device or deliver results in real time, inference speed and model size become critical: k-nearest neighbors can be slow at prediction time, and large ensembles or deep networks may be too heavy. Pruned trees, small neural networks, or logistic regression are preferable in latency-sensitive settings.

The value of ensembles. If initial models do not meet performance requirements, ensembles can boost accuracy. A random forest is a natural next step: it aggregates many trees and often improves substantially over a single tree with little tuning. When random forests underfit, gradient-boosted trees can provide more flexible fits. Be mindful of diminishing returns: complex ensembles are harder to maintain and deploy, and a small gain in accuracy may not justify the added complexity and monitoring overhead.

Rigorous evaluation and tuning. Regardless of which algorithm you choose, use cross-validation to estimate out-of-sample performance and tune hyperparameters. Cross-

validation reveals overfitting: if training performance far exceeds validation performance, you likely need to simplify the model (prune a tree, add regularization, reduce network size). Conversely, if a linear model consistently underfits and leaves systematic patterns in the residuals, you might apply feature transformations or switch to a more flexible algorithm.

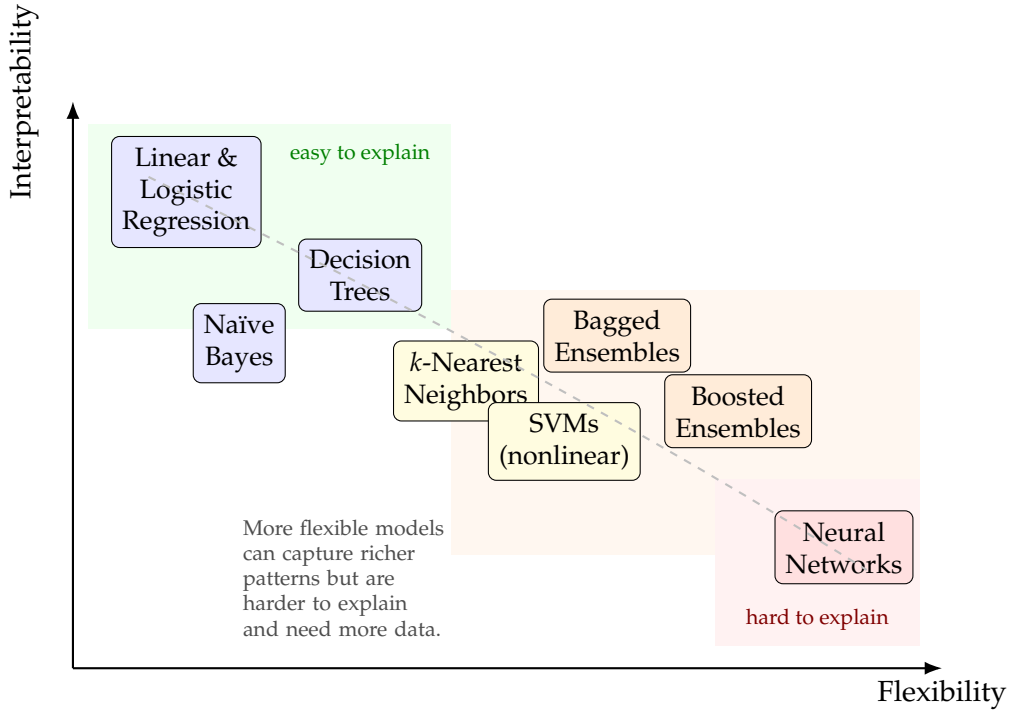


Figure 10.5: The interpretability–flexibility trade-off. Linear and logistic regression sit in the high-interpretability / low-flexibility region; decision trees are interpretable but become more flexible (and harder to explain) as depth increases; k-nearest neighbors and nonlinear SVMs are more flexible yet harder to interpret; ensemble methods such as random forests and gradient boosting further increase flexibility at the cost of interpretability; neural networks are typically the most flexible and least interpretable. The dashed line indicates the general trend: gaining flexibility usually means sacrificing interpretability—although interpretability tools (Section 10.1) can partially close this gap for complex models.

Prefer simplicity. A good rule of thumb is to start simple. A model that meets your accuracy requirements while being easy to implement, explain, and maintain is often better than a marginally more accurate but complex alternative. If logistic regression delivers

an acceptable performance for a marketing prediction, deploying a random forest that improves the metric by one percentage point may not be warranted—complex models are more sensitive to data drift and require more monitoring.

The interpretability–flexibility trade-off. Ultimately, model selection is about balancing interpretability and flexibility, bearing in mind that more flexible models generally require larger datasets. Figure 10.5 illustrates how the methods we have covered stack along these dimensions.

Start with simple methods to establish a baseline and build intuition, advance to more flexible models if necessary, and combine or regularize them when complexity creates problems. By understanding the strengths, weaknesses, and assumptions of different approaches, you can make informed choices and avoid algorithms that are ill-suited to your problem.

Part II

Unsupervised Learning

Introduction to Unsupervised Learning

Unsupervised learning is a paradigm of machine learning when we have no explicit labels or targets. The goal is to discover patterns, groupings, or structure inherent in the data. For example, without knowing which customers belong to which segment (no labels like "high-value" or "budget" customer given upfront), unsupervised algorithms can group customers into segments based on purchasing behavior. These segments might then be interpreted and given business meaning (e.g. "savvy bargain hunters" or "loyal heavy spenders"). More generally, unsupervised methods seek to answer questions such as: Are there natural clusters or segments in the data? Can we reduce data complexity while losing minimal information? Are there unusual data points (i.e., anomalies) that deviate from the norm?

We will focus on three main areas of unsupervised learning:

- **Clustering:** Automatically grouping data into clusters of similar characteristics.
- **Dimensionality Reduction:** Compressing or projecting data into a lower-dimensional form, preserving key structure. Among other uses, this enables us to visualize datasets with many features in two or three dimensions.
- **Autoencoders (Representation Learning):** Neural network-based approaches to learn efficient codings of data.

We will discuss each of these and discuss use cases illustrating their value.

The data preprocessing steps discussed in Section 2—handling missing values, encoding categorical variables, and especially feature scaling—apply to unsupervised learning just as they do to supervised learning. Feature scaling is particularly important here: most unsupervised methods rely on distances or variances, so a feature measured in dollars can easily dominate one measured in units simply because of its scale, producing misleading results.

11 Clustering

Clustering is an unsupervised method for finding useful groupings in data when you do *not* have labels. The goal is to partition observations (e.g., customers, products, transactions, documents, etc) into clusters so that items in the same cluster are more similar to each other than to items in other clusters. This is often used for segmentation: "Which customers behave similarly, and how should we treat them differently?"

Two warnings are due upfront. First, clustering does not discover a single ground-truth answer; different algorithms (and even different runs of the same algorithm) can produce different groupings. Second, clusters are only valuable if they lead to clearer decisions; e.g., clusters that are interpretable and predict different outcomes (churn, conversion, profitability) are more actionable than clusters that merely look separated on a plot.

In this section we cover three common clustering approaches, *K-means*, *Hierarchical clustering*, and *DBSCAN*. Each takes a different approach to defining what a "cluster" is:

- **K-means** assumes clusters are roughly spherical clouds of points in the feature space and tries to find a set of K cluster centers and assign points to the nearest center.
- **Hierarchical clustering** builds a tree of clusters, either agglomerating small clusters into bigger ones or dividing big clusters into smaller ones, yielding a multilevel hierarchy.
- **Density-Based clustering** defines clusters as regions of high density separated by regions of low density; it can find clusters of arbitrary shape and identify outliers as noise.

Each of these methods takes a different approach to defining what a "cluster" is, and we will explain them in turn.

How to judge whether a clustering is "good"? Because clustering is unsupervised, we usually do not have a single correct answer to compare against. Instead, we combine three checks:

i. *Stability:* Do we get similar clusters if we rerun the algorithm with different starting points or on a different sample of the data? If the segments reshuffle dramatically each time, they are unlikely to reflect real structure—they may be artifacts of the algorithm's randomness or of noise in the data.

ii. *Separation and cohesion:* Are clusters reasonably distinct from one another and internally consistent? A useful quantitative tool here is the *silhouette score*, which measures, for each observation, how well it fits its own cluster relative to the nearest alternative cluster (Rousseeuw, 1987). Scores range from $+1$ (the point is firmly inside its cluster) to -1 (the point would fit better elsewhere). A high average silhouette suggests well-separated clusters; a low or negative score suggests the boundaries are blurry. We will also use the *elbow plot*, which shows how cluster tightness improves as the number of clusters increases—a sharp bend (the "elbow") signals the point of diminishing returns. Both tools are demonstrated in the next section.

iii. *Business usefulness:* Can you assign a meaningful interpretation to each cluster, and does that interpretation matter for the decision at hand? Clustering algorithms will always produce groups, but the groups are only valuable if you can describe what makes each one distinctive (e.g., "these customers have short tenure, high monthly charges, and no add-on services") and if those distinctions suggest different actions. A cluster profile heatmap—showing the average value of each feature within each cluster—is the main tool for this step. If you cannot name the clusters in terms that a manager would recognize, or if the profiles do not suggest different pricing, messaging, or service decisions, the clustering may be technically valid but operationally useless.

In practice, the third check is often the most important: clusters are a means to better decisions, not an end in themselves. The next sections introduce specific clustering algorithms along with the visualization needed to apply these three checks.

11.1 K-Means Clustering

K-means is a classic clustering algorithm that aims to partition the dataset into K predefined distinct clusters (Lloyd, 1982). "K" is a number you choose in advance; e.g., segmenting customers into $K = 3$ groups. The "means" refers to representing each cluster by the mean of its points, also known as the cluster *centroid*. The algorithm works iteratively to assign points and adjust the centroids:

- **Initialization:** Start by randomly choosing K initial cluster centers.
- **Assignment step:** Assign each data point to the nearest cluster center (in terms of distance). This forms K clusters based on the current centers.
- **Update step:** For each cluster, recompute the center as the mean of all points assigned to that cluster.
- **Iterate:** Repeat the assignment and update steps until cluster assignments stop changing or changes become very small.

This procedure is trying to make clusters *internally tight*: points assigned to the same cluster should be close to their cluster's centroid. Formally, K-means is trying to minimize the total *within-cluster sum of squares*: the sum, across all clusters, of the squared distances between each point and its cluster's centroid.[14] In plain terms: the algorithm tries to make each cluster as internally tight as possible—points in the same cluster should be close to the cluster's center.

[14] Mathematically, K-means partitions data points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ into K sets $S_1, \ldots, S_K$ to minimize $\sum_{k=1}^{K} \sum_{i \in S_k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2$, where $\boldsymbol{\mu}_k$ is the mean of points in cluster S_k.

Intuitively, K-means searches for K representative "centers" and assigns each point to the nearest center, iterating, until the assignments stabilize. This procedure can sometimes get stuck in a suboptimal grouping (because the initial random centers might lead it astray). Therefore, it is common to run K-means multiple times with different random initial centers and choose the best result (i.e., the one with the lowest sum-of-squares), or to use smarter initialization methods (e.g., pick initial centers that are far apart) to improve clustering quality.

Working Example: Customer Churn. To see K-means in action, we apply it to a customer churn dataset. Each customer is described by features such as tenure (how long they have been a customer), whether they subscribe to internet or phone service, which add-ons they use (online security, tech support, streaming, etc.), contract type, and monthly charges. All features were standardized before clustering so that no single variable dominates the distance calculations (see Section 2.6).

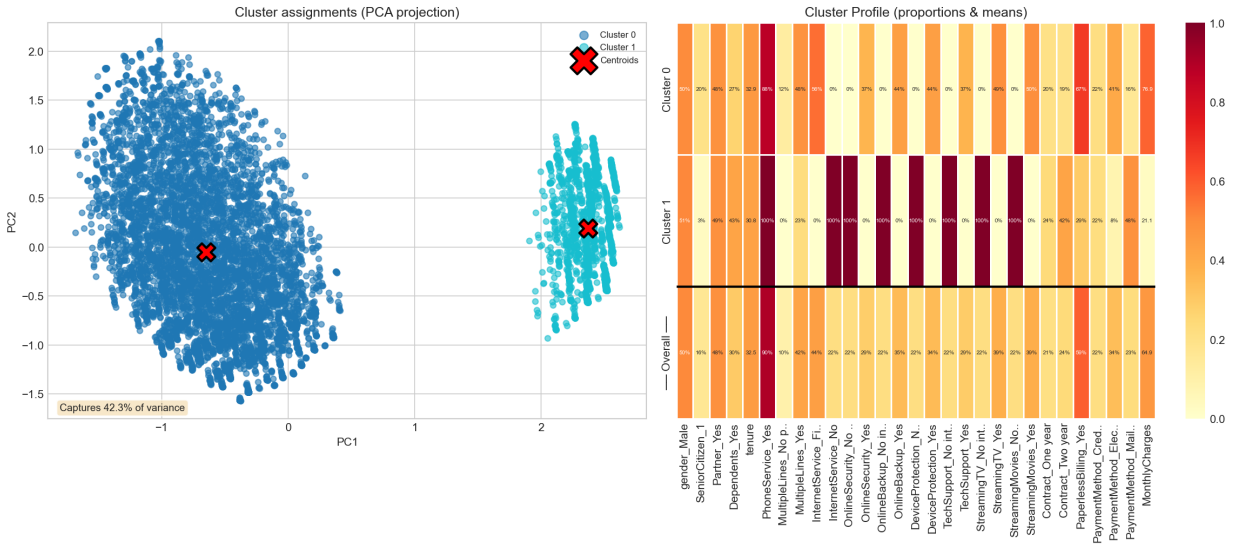


Figure 11.1: **K-means with $K = 2$ on the customer churn dataset.** *Left:* each customer plotted in two summary dimensions (produced by a technique called PCA, covered in Section 12), colored by cluster assignment. *Right: cluster profile heatmap* showing the proportion (for binary features) or mean (for numeric features) of each variable within each cluster and overall. Darker shading indicates higher values.

Figure 11.1 shows the result of K-means with $K = 2$. The left panel plots each customer as a dot in two dimensions. Because the original data has many more than two features, we use a visualization technique called *Principal Component Analysis* (PCA) to compress the data into two summary axes for plotting—we will cover PCA in detail in Section 12. For now, the key point is that the two axes are not individual features; they are summaries that together capture about 42% of the variation in the data, so the picture is useful but incomplete. Two clusters are visible: a large, diffuse group (Cluster 0) and a small, tight group (Cluster 1).

The scatter plot tells us that two groups exist, but it does not tell us *what makes them different*. For that, we turn to the right panel: the **cluster profile heatmap**. This is typically the most useful output of a clustering analysis, because it shows what distinguishes each cluster in terms of the original features.

The profile heatmap in Figure 11.1 identifies two clusters:

- **Cluster 1**—the small, tight group on the right, which consists of customers with no internet service and no phone service—the "no internet" columns are at 100% and all internet add-on features (online security, streaming, device protection, etc.) are at 0%. Their average monthly charge is about $21. These are minimal-service, low-revenue customers.

- **Cluster 0**—the large, diffuse group, which contains everyone else: customers who subscribe to internet service, often with various add-ons, and who pay substantially higher monthly charges (nearly $77).

In short, K-means with $K = 2$ has split the customer base along its most prominent dividing line: customers who use internet services versus those who do not.

Connecting clusters to outcomes. Clustering is unsupervised—we did not use the churn label as an input. But we can still check whether the discovered segments differ in their

churn behavior. In this case, the internet-service cluster (Cluster 0) contains 31.7% churners, while the low-revenue cluster (Cluster 1) contains only 7.2% churners. The gap is striking: customers with internet service churn at over four times the rate of those without it.

This illustrates an important principle: unsupervised learning can serve as a building block for supervised learning. The segments found by clustering—without any knowledge of churn—turn out to differ sharply in churn risk. A manager might use this to prioritize retention efforts on the higher-risk segment, or a data scientist might use cluster membership as an input feature in a downstream churn prediction model. More generally, when labeled data is scarce or expensive, clustering can surface structure that later informs supervised learning.

Choosing K. K-means requires choosing the number of clusters in advance, which is often the hardest part. Two common diagnostic tools can help.

- The **elbow plot** (left panel of Figure 11.2) shows the total within-cluster sum of squares (WCSS) as a function of K. WCSS always decreases as K increases—with enough clusters, every point can be its own cluster—so the question is where the curve *bends*: the point after which adding more clusters brings diminishing returns. In Figure 11.2, the sharpest bend occurs at $K = 2$: going from one cluster to two produces a large drop in WCSS, but subsequent increases in K yield progressively smaller improvements.

- The **silhouette score** (right panel) measures, for each observation, how similar it is to its own cluster compared with the nearest other cluster. Values range from -1 (badly misclassified) to $+1$ (firmly inside its cluster). The average silhouette across all points is plotted for each K; the peak suggests the K with the cleanest separation. Here the peak is at $K = 2$, consistent with the elbow plot.

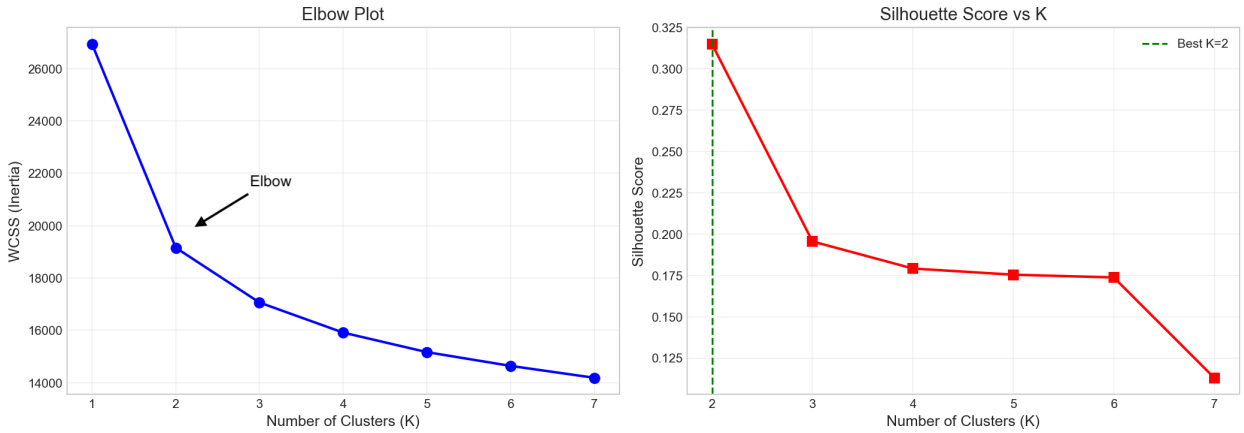


Figure 11.2: **Choosing K for the customer churn dataset.** *Left:* the elbow plot shows a sharp bend at $K = 2$, after which additional clusters bring diminishing reductions in WCSS. *Right:* the silhouette score peaks at $K = 2$ and declines steadily, confirming that the strongest single split is into two groups.

In this dataset, both tools point to $K = 2$: the internet-service vs. no-internet split we saw in Figure 11.1. But this does not mean that $K = 2$ is the "right answer" and all other values are wrong. Elbow and silhouette plots identify the *statistically* most distinct partition, which is not always the most *managerially* useful one. For instance, splitting the internet-service customers further (e.g., into fiber vs. DSL users, or short-tenure vs. long-tenure subscribers) might reveal segments that respond differently to retention offers, even if those segments are not perfectly separated in feature space. In practice, the best approach is to combine these diagnostic plots with business judgment: try several values of K, profile each set of clusters, and ask *"If I had these segments, would I actually change pricing, messaging, or service levels?"* If the answer is no, a different K or a different feature set is usually more productive than a more complicated scoring rule.

Use Cases. Retailers often have massive databases of customer transactions, and clustering can uncover distinct behavioral segments without any pre-labeled categories. A typical setup is to represent each customer with features like purchase frequency, product mix (categories), average basket size, discount sensitivity, and channel usage (online vs. in-store), and then cluster customers into groups that behave similarly.

A well-known example involves Target, the U.S. retailer, whose analysts discovered—while mining purchase data for coupon targeting—that certain product combinations (such as unscented lotion and specific vitamin supplements) reliably preceded purchases of baby-related items (Hill, 2012; Duhigg, 2012). The pattern was not something anyone set out to find; it fell out of a routine segmentation exercise, yet it was accurate enough to flag likely pregnancies before any explicitly baby-related products appeared in a customer's cart. The episode illustrates two lessons. First, behavioral data can be surprisingly revealing—clustering can surface patterns that no one anticipated. Second, segmentation is not just a technical exercise: insights that touch on sensitive personal information can create reputational, legal, and regulatory risk if the practice is perceived as invasive.

K-means has also been used in banking to cluster clients by behavior to detect groups like "credit risk customers" vs "savvy investors", and in healthcare to cluster patients by symptoms or genetic profiles.

Limitations. Because K-means uses means, it is sensitive to outliers—an outlier can drag a centroid toward it. Also, if clusters have very different sizes or densities, K-means can sometimes produce poor results; e.g., it might chop a large diffuse cluster into pieces or ignore a tiny cluster. It also assumes Euclidean distance is meaningful for your features; if features are on very different scales, proper normalization is needed (Section 2.6). Despite these limitations, K-means often provides a quick and reasonable partition and is easy to implement.

Gaussian Mixture Models: A Flexible Extension of K-Means. K-means assigns each observation to exactly one cluster based on which centroid is closest. This "hard" assignment can be misleading when a data point sits near the boundary between two segments—K-means puts them firmly in one group even though they resemble both almost equally.

Gaussian Mixture Models (GMMs) relax this assumption. Instead of representing each cluster as a single point (the centroid), a GMM represents each cluster as a bell-curve-shaped cloud with its own center, spread, and shape. The algorithm then estimates, for each observation, the probability that it belongs to each cluster. A customer might be assigned 70% probability of belonging to the "price-sensitive" segment and 30% probability of belonging to the "convenience-oriented" segment. This *soft assignment* provides a richer picture than a binary label and can be useful when segment boundaries are blurry—as they often are in practice.

GMMs also handle cluster shapes that K-means cannot. Because each cloud has its own spread and orientation, GMMs can capture elongated or elliptical clusters, whereas K-means implicitly assumes all clusters are roughly spherical and similarly sized.

Like K-means, GMMs require choosing the number of clusters in advance. However, GMMs are more computationally expensive and can be unstable when the dataset is small relative to the number of features, because estimating the shape of each cloud requires enough data to pin down its spread in every direction. They also assume that the data within each cluster follows a roughly bell-curved distribution, which is not always the case. For most egmentation problems, K-means is the practical starting point; GMMs are worth considering when probabilistic segment membership is needed.

11.2 Hierarchical Clustering

Hierarchical clustering builds a multi-level hierarchy of clusters. The result is often visualized as a tree diagram called a dendrogram, which shows how clusters merge or split at different distance thresholds. The most common method is *agglomerative (bottom-up) clustering*, which starts with each data point as its own cluster (e.g., n clusters for n points); then, iteratively merges the two "closest" clusters until we end up with one big cluster

containing everything.[15] The procedure for agglomerative clustering operates iteratively as follows:

- **Initialization:** Each point starts as its own cluster.
- **Compute distances between clusters:** At each step, the algorithm quantifies how "close" two clusters are. At the initial step when each data point is its own cluster, this is just the distance between pairs of points (e.g., Euclidean distance after standardizing features). After clusters start merging, we compute distances between *clusters* using a linkage rule (described below).
- **Merge clusters:** Find the two clusters that are closest under the chosen linkage rule, and merge them into a single larger cluster.
- **Iterate:** Continue the process of recomputing the distance matrix and merging the closest pairs of clusters until all points are merged into one cluster.

The key choice in hierarchical clustering is the definition of inter-cluster distance—how do we measure the distance between two clusters? Common linkage criteria are:

- **Single linkage** which takes the distance between two clusters to be the smallest distance between any single point in one cluster and any point in the other cluster. In other words, clusters are as close as their two most similar members. This tends to produce long, "chain" clusters (due to a "friends-of-friends" effect).
- **Complete linkage** which takes the distance between two clusters to be the largest distance between any point in one cluster and any point in the other. Clusters must be close in the sense that all members are within some maximum bound. This tends to produce more compact, spherical clusters.

[15] Another method is divisive (top-down) clustering which starts with all points in one cluster, and recursively splits clusters into two until each point is alone. This is less commonly used, in part because it's computationally more expensive.

- **Average linkage** which takes the distance to be the average distance between all pairs of points (one from each cluster).
- **Ward linkage** which merges the pair of clusters that causes the smallest increase in within-cluster sum of squares (variance). This tends to produce compact, well-separated clusters, especially for numeric data.

For most business applications with numeric data, Ward linkage is a sensible default because it tends to produce compact, similarly-sized clusters. Single linkage is worth trying when you suspect clusters may be elongated or chain-like, but it is more sensitive to noise.

Unlike K-means, agglomerative clustering does not require specifying K upfront. Instead, one can "cut" the dendrogram at a chosen level to get a desired number of clusters. This flexibility is a key advantage: you produce a whole hierarchy and can decide later whether you want 2 clusters or 10, by choosing the appropriate cut level.

Working Example: Customer Churn (continued). We apply hierarchical clustering with Ward linkage to the same customer churn dataset. Figure 11.3 shows the results for two cut levels: $K = 2$ (top row) and $K = 3$ (bottom row).

With $K = 2$ (top row), the result is nearly identical to what K-means produced in Figure 11.1: Cluster 1 is the small, tight no-internet group, and Cluster 0 is the broad internet-service group. This is reassuring—when two different algorithms converge on the same split, it suggests the structure is genuine rather than an artifact of one particular method.

The $K = 3$ cut (bottom row) is where hierarchical clustering shows its value. The no-internet cluster (Cluster 1) remains unchanged—it was already tight and homogeneous, so the algorithm sees no reason to split it. Instead, the internet-service group is subdivided into two segments. Reading the profile heatmap: Cluster 0 consists of customers with high monthly charges (about \$87 on average), a mix of add-on services, and longer contracts, while Cluster 2 has lower monthly charges and fewer add-ons. This is the kind

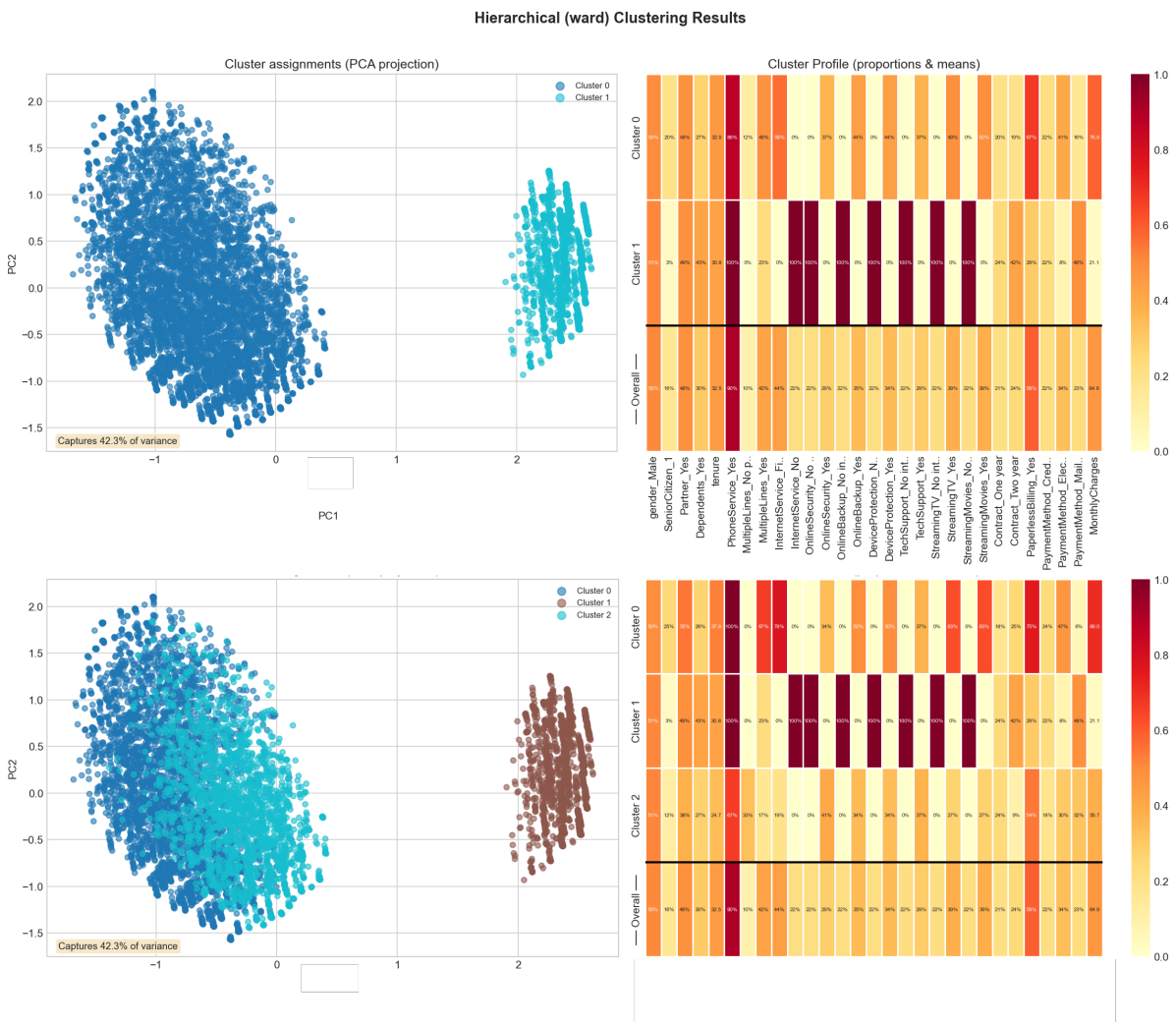


Figure 11.3: **Hierarchical clustering (Ward linkage) on the customer churn dataset.** Top row: cutting the hierarchy at $K = 2$ produces essentially the same split as K-means—an internet-service cluster and a no-internet cluster. Bottom row: cutting at $K = 3$ keeps the no-internet cluster intact and subdivides the internet-service customers into two groups. Left panels show cluster assignments projected onto two summary dimensions; right panels show the cluster profile heatmap for each cut.

of progressive refinement that hierarchical clustering is designed for: the first split captures the dominant divide in the data, and subsequent splits reveal finer structure within the larger group.

The dendrogram in Figure 11.4 visualizes the full merging history:

- **Vertical axis (distance):** Each horizontal line connecting two branches represents a merge, and its height indicates the distance at which that merge occurred. Merges

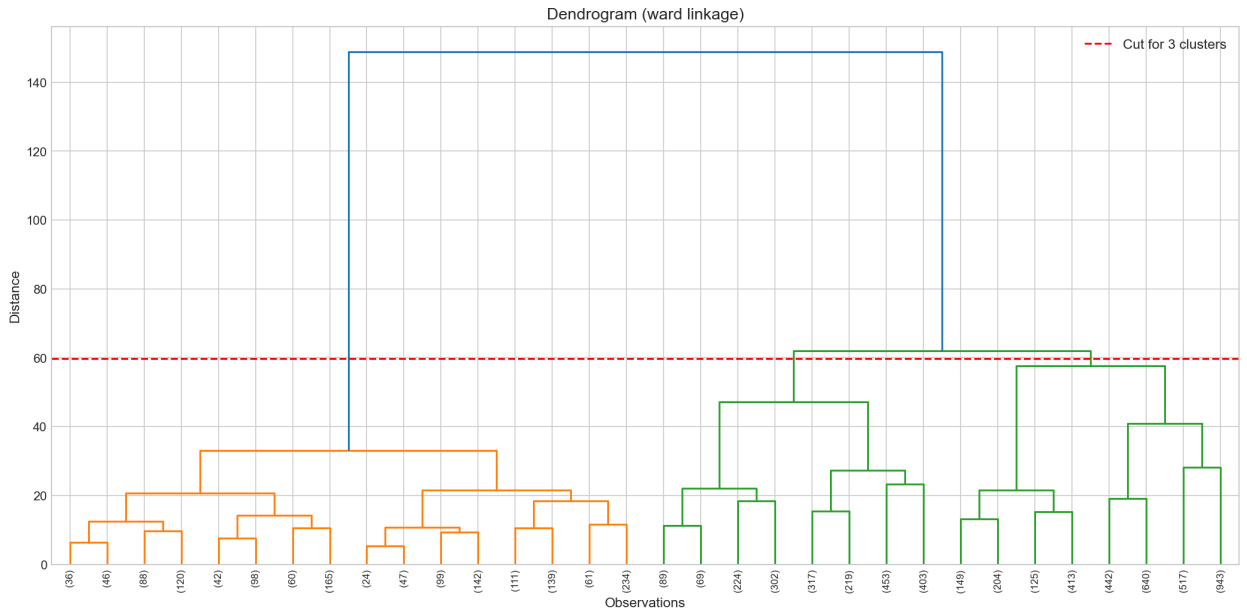


Figure 11.4: **Dendrogram for the customer churn dataset (Ward linkage, truncated to 5 levels).** Each leaf at the bottom represents a group of observations; the number in parentheses is the group size. The vertical axis measures the distance (cost) at which two groups were merged—higher merges mean the groups being joined were less similar. The red dashed line shows the cut at distance ≈ 60 that yields 3 clusters. The final merge (blue, at distance ≈ 148) joins the no-internet cluster (orange branches, left) with the internet-service cluster (green branches, right). The large jump from ≈ 60 to ≈ 148 confirms that the 2-cluster split is the strongest structure in the data.

near the bottom of the tree join very similar groups; merges near the top join dissimilar groups.

- **Leaves and cluster sizes:** The tree is truncated to 5 levels for readability. Each leaf at the bottom represents a group of observations rather than an individual point; the number in parentheses is the group size (e.g., "(943)" at the far right means a sub-cluster of 943 customers).

- **The red dashed line ($K = 3$ cut):** Drawing a horizontal line at distance ≈ 60 intersects three branches, yielding three clusters. Everything below the line within a single branch belongs to the same cluster. The orange branches on the left form one cluster (the no-internet customers), and the green branches on the right form two clusters (two types of internet-service customers).

- **The big jump:** The most informative feature of a dendrogram is often a large gap between successive merge heights. Here, the sub-clusters within each group merge at distances below 60, but the final merge joining the two main branches (the blue line) occurs at distance ≈ 150—more than double. This large jump tells us that the 2-cluster split is by far the dominant structure: the two groups are internally cohesive but very different from each other. This is consistent with the elbow and silhouette analysis from the K-means section.

The general rule for reading dendrograms is to look for large jumps in merge height. A big jump suggests a "natural" boundary—clusters below that height are reasonably tight, but merging them further requires bridging a large gap. Where there is no obvious jump (the tree rises gradually), the data does not have a single clear-cut number of clusters, and business judgment must guide the choice.

Use Cases. Hierarchical clustering is often used in genomics and bioinformatics to cluster genes or patient samples. For instance, in gene expression analysis, dendrograms can show how samples cluster (perhaps indicating subtypes of a disease) and how genes cluster (perhaps indicating functional groups).

In marketing analytics, firms use hierarchical clustering to first divide customers broadly and then find subsegments; this is useful when a natural taxonomy is suspected. The churn example above illustrates this: the first cut separates internet from non-internet customers, and finer cuts reveal sub-segments within the internet group that might warrant different retention strategies.

Computational Note: Hierarchical clustering is more computationally intensive than K-means for large datasets. Very large datasets (millions of data points) may be infeasible to cluster hierarchically without special techniques; K-means or other methods are often preferred at large scale.

Linkage Effects: Single linkage can sometimes yield one giant cluster and many singletons. Complete linkage is more stringent and often more balanced but can break large natural clusters if there is even a single outlier far from the rest. Practitioners often experiment with different linkage methods or use domain knowledge to choose one.

In summary, hierarchical clustering is especially useful when a nested hierarchy of segments—a tree that can be cut at different levels—is desired.

11.3 Density-Based Clustering

Density-based clustering takes a fundamentally different approach. Instead of partitioning data into a pre-specified number of groups, it defines clusters as regions where data points are packed closely together, separated by sparser areas. Points in sparse regions are labeled as *noise* and left unclustered (Ester et al., 1996).

This difference has three practical consequences. First, density-based methods do not require you to choose the number of clusters in advance—the algorithm discovers however many dense groups the data naturally contains. Second, clusters can take on arbitrary shapes, not just the roughly spherical blobs that K-means tends to produce. Third—and most important for many business applications—the algorithm explicitly identifies outliers rather than forcing every observation into a cluster. In fraud detection, for example, those outlier points *are* the suspicious transactions—the object of interest.

Key Concepts. The ideas behind density-based clustering are best understood through three types of points: A *core point* is a point that has at least some minimum number of neighbors—dictated by a parameter of the model—within a specified radius. These are points in the "thick" of a cluster. A *border point* falls within the radius of a core point but does not itself have enough neighbors to be a core point. It sits on the edge of a cluster. A *noise point* is not within reach of any core point. It does not belong to any cluster.

Clusters are formed by linking core points that are neighbors of one another—if you

can walk from one core point to another through a chain of nearby core points, they belong to the same cluster. Border points are assigned to the cluster of the nearest core point they touch. Noise points are left unassigned.

HDBSCAN

HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) is the most widely used density-based clustering algorithm in practice (Campello, Moulavi and Sander, 2013). Its key idea is to examine the data at multiple scales of density simultaneously. Imagine slowly lowering the threshold for what counts as "dense": at first, only the very tightest pockets of points qualify as clusters; as the threshold relaxes, broader groupings emerge and eventually everything merges into one mass. HDBSCAN builds this entire hierarchy and then asks: which clusters are *stable*—that is, which groupings persist over a wide range of density thresholds rather than appearing briefly and dissolving? Stable clusters are retained; fleeting ones are discarded.

This multi-scale approach has two important consequences. First, it can find tight clusters and diffuse clusters in the same dataset, because each cluster is evaluated at the density scale where it is most coherent—there is no single threshold that must work for all groups at once. Second, it discovers the number of clusters from the data rather than requiring the analyst to specify it in advance. Points that do not belong to any stable dense region at any scale are labeled as noise.

HDBSCAN also outputs a *membership probability* for each point, indicating how confidently it belongs to its assigned cluster. A probability near 1 means the point sits firmly inside a dense region; a lower probability means it is near a cluster boundary. This is useful when cluster edges are fuzzy—rather than a hard yes-or-no assignment, the algorithm gives a measure of certainty that can inform downstream decisions (e.g., routing borderline transactions to manual review).

Preprocessing for HDBSCAN. Because HDBSCAN relies on distances between points to determine density, preprocessing choices can substantially affect the results.

Feature scaling is essential—if one feature ranges from 0 to 100,000 (e.g., transaction amount in cents) and another from 0 to 1 (e.g., a binary indicator for card-present), the distance calculations will be dominated by the first feature.

Outlier handling requires a lighter touch than with K-means or hierarchical clustering. With those methods, extreme outliers can drag centroids or warp dendrograms, so it is common to winsorize or remove them before clustering. With HDBSCAN, outliers are part of the point: the algorithm is designed to label them as noise rather than force them into clusters. Aggressively removing outliers before running HDBSCAN risks discarding precisely the observations the method is built to surface. That said, obvious data-entry errors should still be corrected or removed, because they are not genuine anomalies—they are corrupted data.

Dimensionality can also matter. HDBSCAN measures density in the full feature space, and in high-dimensional spaces distances between points tend to become more uniform (a phenomenon known as the curse of dimensionality), making it harder to distinguish dense regions from sparse ones. If you have many features—especially many one-hot encoded categorical variables—consider reducing dimensionality first (e.g., via PCA, covered in Section 12) before running HDBSCAN.

Choosing parameters. HDBSCAN has one main parameter: `min_cluster_size`, which is the smallest group of points that the algorithm will consider a "real" cluster rather than noise. This choice should be driven from domain knowledge: for example, if you are segmenting a customer base of 10,000, you might set `min_cluster_size` to 50 because a segment of fewer than 50 customers is too small to act on.

A secondary parameter, `min_samples`, controls how conservative the algorithm is about labeling points as noise. Higher values require denser neighborhoods to qualify as core

points, which means more points get classified as noise. A practical starting point is to set `min_samples` to `min_cluster_size` or slightly smaller, and then adjust it downwards if too many points are labeled as noise.

Parameter change	Number of clusters	# Noise points
`min_cluster_size` ↑	*Decreases*: small groups no longer qualify as clusters and are reclassified as noise	*Increase*: former small clusters become noise
`min_samples` ↓	*Tends to increase*: looser density requirement allows more clusters to form	*Decreases*: more points pass the density test and are assigned to clusters

Table 5: How HDBSCAN's two main parameters affect the clustering output.

Working Example: Credit-Card Fraud Detection. To see HDBSCAN in action, we apply it to a synthetic credit card fraud dataset. Each transaction is described by features such as merchant location, merchant risk score, distance from the cardholder's home, whether the card was physically present, whether the transaction was cross-border, transaction amount, and short-run transaction velocity (how many transactions occurred in the preceding 24 hours). The data contain three underlying patterns: a large mass of routine transactions, a small dense burst of coordinated suspicious activity, and a scattering of isolated fraud-like anomalies.

Figure 11.5 shows the results with `min_cluster_size` = 200 (the smallest group we consider a real segment) and `min_samples` = 15. HDBSCAN produces three groups, and each tells a different story:

- **Cluster 1** ($n = 18{,}852$) contains virtually all routine transactions. Its profile is low-risk and local: merchant risk is below average, distance from home is low, card-present transactions are common, cross-border transactions are rare, and both amount and velocity are near the overall mean. This is a data-driven definition of what typical spending looks like.

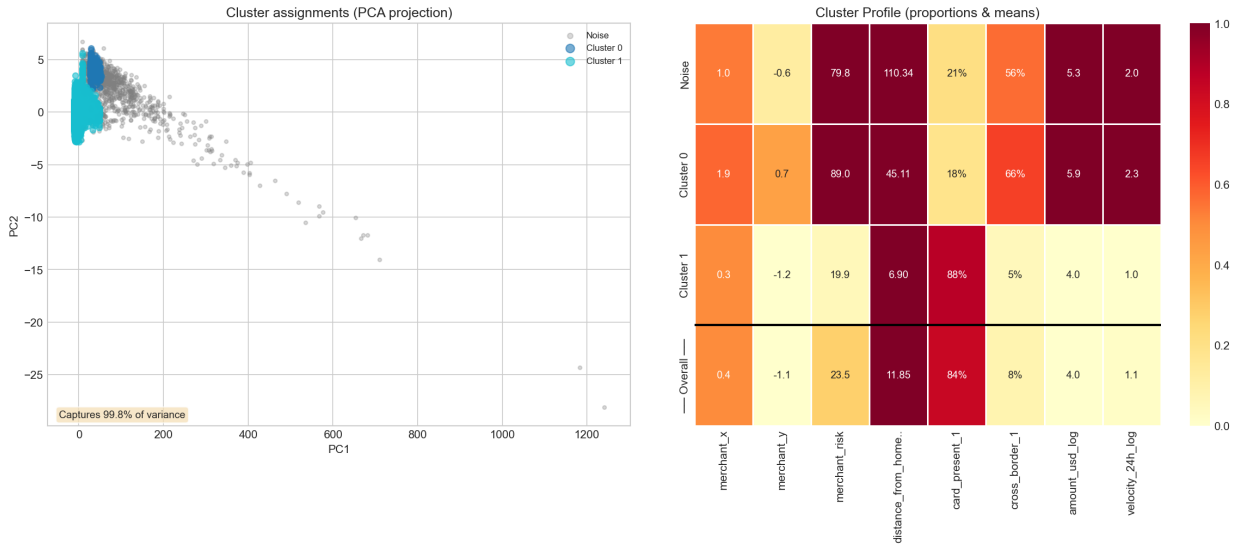


Figure 11.5: **HDBSCAN on credit card transaction data.** Top left: each transaction plotted in two PCA dimensions—noise points (gray) are scattered far from the main mass. Top right: cluster profile heatmap showing the mean of each feature within each cluster and the noise set. Bottom left: cluster sizes—one large normal cluster, one small suspicious cluster, and 844 noise points. Bottom right: HDBSCAN membership probability distribution—most clustered points have probability near 1, indicating high-confidence assignments.

- **Cluster 0** ($n = 304$) is smaller but informative. Its profile is highly suspicious: merchant risk is very high (89.0 vs. 23.5 overall), distance from home is far above average (45 vs. 12), card-present transactions are rare (18%), cross-border transactions are frequent (66% vs. 8% overall), and transaction velocity is elevated. HDBSCAN has extracted a compact pocket of coordinated abnormal activity as its own cluster.

- **Noise** ($n = 844$) consists of isolated or weakly connected observations that do not belong to any stable dense region. These are not random failures of the algorithm—their profile is even more extreme than Cluster 0 on some dimensions, especially distance from home (110) and cross-border activity (56%). The correct interpretation is that HDBSCAN is distinguishing between two kinds of suspicious behavior: a dense pocket regular enough to deserve cluster status, and a set of scattered anomalies that warrant individual investigation.

This three-way decomposition—stable normal core, compact suspicious cluster, and irregular suspicious fringe—is precisely the kind of output that makes HDBSCAN valuable for fraud detection. The algorithm is not claiming that all noise points are fraudulent, nor that all fraud must be noise. It is separating the data into dense behavioral regimes and a set of observations unusual enough to warrant further attention. A manager receiving this output has two immediate actions: (i) use the dominant cluster's profile as a baseline for monitoring (transactions that deviate from this profile merit a closer look), and (ii) route the small cluster and the noise points to a fraud investigation team for review.

Use Cases. Density-based clustering has proved most valuable in areas where automatic cluster discovery, arbitrary cluster shapes, and built-in noise detection provide a clear advantage over simpler methods.

Fraud and anomaly detection. The ability to label low-density points as noise makes density-based methods a natural fit for fraud detection, where the goal is precisely to find observations that do not belong to any normal pattern. Financial institutions use these methods to cluster typical transaction behavior and flag isolated, irregular transactions as potential fraud. Databricks Labs built an open-source tool called GEOSCAN—a DBSCAN variant designed for credit card fraud detection at scale—that clusters each customer's normal transaction locations and flags purchases outside those zones in near-real-time (Databricks Labs, 2021). Morocco's central bank (Bank Al-Maghrib) deployed an ensemble model including HDBSCAN to detect anomalies in financial statements submitted by regulated institutions—errors, fraud, or systemic risks that could distort monetary policy decisions (Bank for International Settlements, 2024). In insurance, the actuarial firm Milliman benchmarked several anomaly detection methods and found that HDBSCAN achieved complete detection of all ground-truth anomalies on appropriately sized datasets, though it noted that computational cost can be a constraint on very large

datasets (Milliman, 2025).

Geospatial and logistics applications. Geographic data—GPS coordinates, delivery addresses, store locations—clusters along streets, coastlines, and transit corridors rather than in neat circles. K-means's assumption of spherical clusters produces poor results in this setting, which is why major geospatial platforms (Google BigQuery GIS, PostGIS, Esri's ArcGIS Pro) all include built-in density-based clustering functions.

Ride-hailing company Grab used DBSCAN with a 300-meter radius on booking coordinates (pickup and dropoff locations) to assess the viability of its GrabShare ride-pooling service across Southeast Asia. The analysis revealed that 35–46% of peak-hour bookings fell into matchable clusters—enough to justify launching the service (Grab Engineering, 2018). GOGOX (formerly GoGoVan), a Hong Kong logistics platform, developed a recursive variant of DBSCAN to solve delivery routing problems. A uniform radius failed because dense urban areas like Tsim Sha Tsui might contain 1,000 orders within a 1 km radius while suburban areas contained only a handful. Their recursive approach subdivides dense clusters while grouping sparse ones, achieving a 61% decrease in route optimization time compared to standard methods. The underlying algorithm is described in (GOGOX Technology, 2022; Bujel, Lai and Szczerbicki, 2019).

Text analytics and customer feedback. A newer and rapidly growing application is clustering unstructured text—customer reviews, support tickets, survey responses—after converting text into numerical vectors using embedding models.[16] GitHub's Customer Success team, for example, uses HDBSCAN to automatically categorize customer support tickets into topic clusters, surfacing emerging issues and feature requests that would be invisible in manual review (GitHub Customer Success, 2024). DoorDash applied DBSCAN to financial variance analysis, where one discovered cluster of 216 journal entries

[16]Embedding models map a block of text into a numeric vector designed to capture meaning. Conceptually, text-embedding models try to ensure that two blocks of text that are semantically similar are close to each other in vector space, and vice versa.

all related to order cart adjustments—a pattern that was invisible to manual review (DoorDash, 2025).

Text data is particularly well-suited to density-based clustering because the number of topics is unknown in advance, topic clusters are often irregularly shaped in the embedding space, and some documents do not fit neatly into any topic (noise).

11.4 Comparison of Clustering Methods

Each clustering approach has strengths that match different problem characteristics.

K-means is the go-to method for a quick, interpretable partition of the data into a prespecified number of segments. Hierarchical clustering is suitable for exploring a hierarchy of progressively finer or nested segments. HDBSCAN is better suited if the data contains clusters of irregular shape or meaningful outliers—especially if identifying those outliers is key.

A key point to bear in mind is that these methods are complements: it is common practice to run more than one and compare results, treating agreement across methods as evidence that the structure is genuine.

12 Dimensionality Reduction

In many datasets, we have a large number of features (dimensions). However, high-dimensional data can be difficult to work with: It is hard to visualize (we can't directly plot points beyond 3D), many features might be correlated or redundant, and the *curse of dimensionality* (i.e., that high-dimensional spaces can be mostly empty) often makes analysis computationally and statistically challenging.

Dimensionality reduction techniques address this by transforming data from a high-dimensional space to a lower-dimensional space (e.g., compressing 100 features into 2 or 3 synthetic features) while trying to preserve as much important structure as possible. This is analogous to finding the "essence" of the data. It can be used for visualization (i.e., to reduce a high-dimensional dataset to 2D or 3D for plotting), as a pre-processing step to reduce complexity for subsequent modeling (e.g., avoiding multicollinearity or speeding up a supervised learning algorithm), as well as for noise reduction, data compression, and others. We will focus on PCA as the main workhorse for dimensionality reduction and interpretability, and we will briefly discuss a visualization-first technique, t-SNE and its modern cousin, UMAP.

12.1 Principal Component Analysis

PCA is a powerful tool for simplifying complex data. Its goal is to find a new set of summary axes (called principal components) that capture the most variance in the data. Simply put, PCA finds a new set of axes for the data such that the first axis (principal component 1) is the direction along which the data varies the most; the second axis is the direction of next highest variance, subject to being uncorrelated with the first; the third is the next highest variance uncorrelated with both prior directions, and so on. By projecting the data onto the first few principal components, PCA gives a lower-dimensional representation that retains most of the information present in the original data (Jolliffe

and Cadima, 2016).

A Toy Example

Consider data on two features, x_1 and x_2. If the two features are perfectly correlated (i.e., x_2 is a linear function of x_1), then the data points lie on a straight line (Figure 12.1, left panel). In this case, PCA identifies the line as the first principal component (PC_1), which captures all of the variation. The second component (PC_2), perpendicular to the line, carries zero information because the data does not vary in that direction. We started with two variables, but one was redundant, and PCA discovered the redundancy.

Now suppose x_1 and x_2 are correlated but not perfectly; e.g., let x_1 be units sold and x_2 be revenue. The data points form an elongated cloud rather than an exact line (Figure 12.1, right panel). PCA places PC_1 along the long axis of the cloud (the direction of greatest variability) and PC_2 perpendicular to it (the direction of least variability). If PC_1 captures most of the variance, we can drop PC_2 and approximate the data with a single summary variable, effectively filtering out the noise in the perpendicular direction while retaining the main trend.

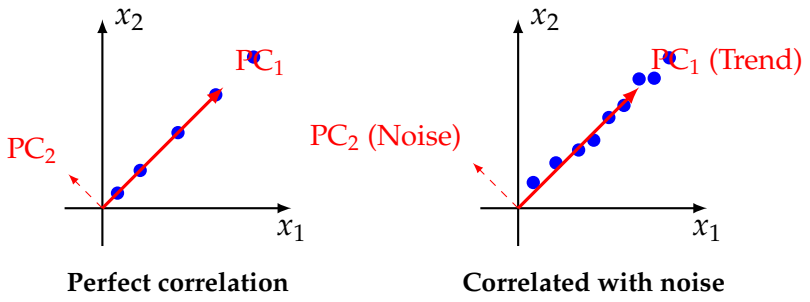


Figure 12.1: **PCA on two variables.** Left: when x_2 is a perfect linear function of x_1, all points lie on a line. PC_1 runs along the line and captures 100% of the variation; PC_2 carries zero. Right: when the variables are correlated but not perfectly, the data forms an elongated cloud. PC_1 captures the main trend; PC_2 captures the smaller residual variation, which is often mostly noise.

This toy example illustrates the core idea: PCA rotates the axes to align with the directions of greatest variation, allowing us to keep the most informative directions and

discard the rest. The same logic extends to any number of features. With 50 features, PCA might find that the first 5 principal components capture 80% of the total variation, meaning the data, despite nominally living in a 50-dimensional space, has only about 5 truly independent dimensions of variation.

What PCA Is (and Isn't) Good For

Before diving into the technical details, it is worth understanding the main reasons practitioners use PCA.

Visualization. With more than three features, data cannot be plotted directly. PCA projects high-dimensional data onto two or three summary axes, producing a scatter plot that reveals patterns, clusters, and outliers. These summary axes are not original features—they are weighted combinations of all features—but they provide the most informative low-dimensional picture possible (in the sense of preserving the most variance).

Compression and noise reduction. Many datasets contain features that are highly correlated with one another and thus partially redundant. PCA identifies this redundancy and replaces many correlated features with a smaller number of uncorrelated summary variables. This can make downstream analyses faster, more stable, and less prone to overfitting.

Preprocessing for supervised learning. PCA is often used as a preliminary step before training a predictive model. Instead of feeding 100 correlated features into a regression or classification algorithm, you can feed the first 10 principal components. This can improve model stability and reduce overfitting. However, there is an important caveat: PCA selects directions based on *variance in the features*, not on *predictive power for the outcome*. A component that captures very little variance overall might still be highly predictive of the target. Discarding it in the name of compression could hurt predictive performance. We return to this point below.

What PCA is <u>not</u> good for. PCA captures only *linear* relationships. If the meaningful

structure in the data is curved or nonlinear, PCA may miss it entirely. Nonlinear alternatives such as t-SNE and UMAP, covered in Section 12.2, are better suited for those situations. PCA also does not identify clusters or segments on its own—it simplifies the data, but you still need a clustering algorithm or visual inspection to find groups. Finally, PCA components are "abstract" axes that mix multiple original features together, which can make them harder to interpret than the raw features themselves. We discuss interpretation strategies below.

Preprocessing for PCA

PCA's results depend heavily on how the data are prepared. Three preprocessing decisions deserve attention.

Feature scaling is essential. PCA works by finding directions of maximum variance. If one feature is measured in dollars (values in the millions) and another in units (values in the single digits), the dollar feature will dominate the variance simply because of its scale, and the first principal component will essentially replicate that one feature.

PCA is designed for continuous (numeric) variables. The linear algebra behind PCA assumes that features are continuous and that the correlation structure among them is meaningful. While it can handle binary (0/1) indicator variables, it works best when most features are numeric.

Alternatives for categorical data. When the data is mostly or entirely categorical, two alternatives are worth considering. *Multiple Correspondence Analysis* (MCA) is the categorical analogue of PCA—it operates on frequency tables rather than correlation matrices and is designed for categorical variables. For mixed data (some continuous, some categorical), *Factor Analysis of Mixed Data* (FAMD) extends PCA to handle both types simultaneously by applying PCA logic to the continuous features and MCA logic to the categorical ones.

In practice, if the data contains a handful of categorical variables among many numeric ones, one-hot encoding followed by standard PCA is usually fine. If the dataset is

predominantly categorical, MCA or FAMD is the better starting point.

PCA Output

Starting with features $x_1, \ldots, x_p$, PCA produces three outputs.

Loadings. Each principal component is defined by a set of *loadings* (also called weights)—one per original feature—that describe how much each feature contributes to that component. For example, PC_1 might put a large positive loading on *sqft_living*, a moderate positive loading on *bathrooms*, and a negative loading on *yrs_since_renovation*. The sign and magnitude of the loadings are what give the component its meaning: they tell you what "mixture" of original features the component represents. Interpreting loadings is the primary way to assign a meaningful label to a component (e.g., "this component represents overall house size and quality").

Explained variance. PCA reports the fraction of total variance captured by each component. PC_1 always explains the most, PC_2 the next most, and so on. A common visualization is the *scree plot* (a bar chart of explained variance per component with a cumulative line), which helps answer: "How many components do I need to retain most of the information?" A common rule of thumb is to keep enough components to reach 80% cumulative variance, though the right threshold depends on the application.

Scores. For each observation, PCA computes a *score* on each component—the observation's coordinate in the new PCA axis system. Each score is a weighted combination of the original features:

$$z_i = w_{i1}\, x_1 + w_{i2}\, x_2 + \cdots + w_{ip}\, x_p.$$

Intuitively, z_1 says "where this observation sits along the main pattern in the data", z_2 says "where it sits along the second-most important pattern", and so on. These scores are the new synthetic features that can be used for visualization (plot z_1 vs. z_2), as inputs to a supervised model, or for clustering.

Often, the first few principal components explain a large fraction of the overall vari-

ance. Based on explained variance or other criteria, we typically keep the first $k < p$ principal components. The resulting scores $(z_1, \ldots, z_k)$ are uncorrelated with each other (which simplifies many downstream models), ordered by importance, and often much fewer in number than the original p features while still capturing most of the variance.

Working Example: House Prices

We apply PCA to a subset of features from the house price dataset from earlier. The features include square footage of living space, lot size, number of bedrooms and bathrooms, construction grade, condition, view quality, year built, years since renovation, number of floors, fraction of area above ground, and a waterfront indicator. All features were scaled before running PCA. Figure 12.2 shows four diagnostic panels.

Panel 1 — Scree plot (top left): "How many dimensions does this data really have?" PC1 captures about 30% of the total variance, PC2 about 16%, PC3 about 10%, PC4 and PC5 about 8% each. The cumulative line reaches roughly 46% after two components and 72% after five. This tells us that the housing data does not compress as neatly as some datasets: there is no single dominant factor, and substantial information lives beyond the first two dimensions. The orange dashed line marks the selection of five components, which is where the individual bars flatten out and each additional component contributes diminishing returns. The practical implication is that any 2D scatter plot of this data captures less than half the variation—it is a useful but decidedly incomplete summary.

Panel 2 — Score plot (top right): "What does the 2D summary look like?" Each dot is a house, plotted along PC1 (30.4% of variance) and PC2 (15.7%), colored by sale price. The main mass of homes forms a single dense cloud of dark purple, because the raw price scale is stretched by a handful of multi-million-dollar properties—nearly all homes fall in a narrow price band relative to those outliers. The most visible pattern is a *rightward tail of extreme observations* beyond PC1 ≈ 5, where a few scattered teal and green dots indicate

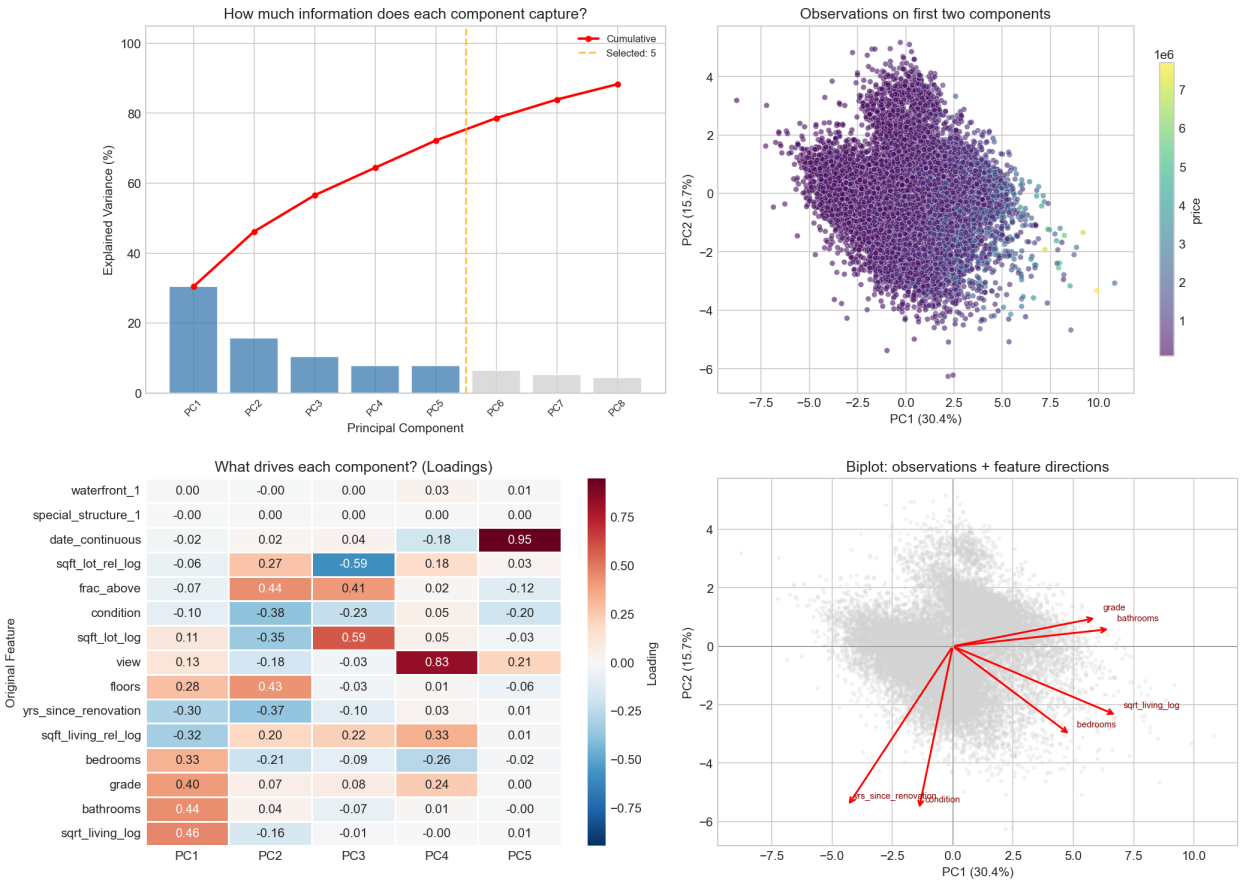


Figure 12.2: **PCA on the house price dataset.** Top left: scree plot showing explained variance per component and the cumulative total. Top right: each house plotted on the first two components, colored by sale price. Bottom left: loadings heatmap showing how each original feature contributes to each component. Bottom right: biplot overlaying feature directions (red arrows) on the score scatter.

the most expensive properties. These are likely the largest, highest-grade luxury homes. Within the main cloud (PC1 from roughly -7 to 3), price differences are not visually distinguishable—the colors are nearly uniform. The vertical spread (PC2) does not track price either: moving up or down in the plot produces no visible color shift, confirming that PC2 captures a dimension of variation largely independent of price.

Note that the data does not form distinct clusters—it is a single dense cloud with a long right tail. PCA is serving here as a compression and interpretation tool, not as a segmentation tool.

Panel 3 — Loadings heatmap (bottom left): "What does each component mean?" This is typically the most useful panel for interpretation. The heatmap shows five components across 15 features. Reading column by column:

- **PC1 — "Size and quality."** The strongest positive loadings are *sqrt_living_log* (0.46), *bathrooms* (0.44), *grade* (0.40), *bedrooms* (0.33), and *floors* (0.28). The strongest negative loadings are *sqft_living_rel_log* (-0.32) and *yrs_since_renovation* (-0.30). A house that scores high on PC1 is large in absolute terms, high-grade, has many bedrooms and bathrooms, multiple floors, was renovated recently, and neighbors even larger homes. This is essentially a "how much house is this?" dimension, and it is why the most expensive properties appear at the far right of the score plot.

- **PC2 — "Building style and lot type."** The positive loadings are *frac_above* (0.44) and *floors* (0.43); the negative loadings are *condition* (-0.38), *yrs_since_renovation* (-0.37), and *sqft_lot_log* (-0.35). A house scoring high on PC2 is a multi-story, recently renovated structure with most of its area above ground, sitting on a smaller lot.

- **PC3 — "Lot size contrast."** This component is driven by a contrast between *sqft_lot_log* (0.59) and *sqft_lot_rel_log* (-0.59), with a secondary positive loading on *frac_above* (0.41). It captures the difference between absolute lot size and lot size relative to the neighborhood. A high PC3 score indicates a property with a large lot in absolute terms but small relative to its neighbors—or equivalently, a property in an area where lots are generally large.

- **PC4 — "View quality."** Dominated by *view* (0.83). This captures whether the property has a view, largely independent of its size, age, or lot.

- **PC5 — "Sale timing."** Dominated by *date_continuous* (0.95). This component is almost entirely about *when* the home was sold. It captures market-wide price trends over the period covered by the dataset.

Two features have near-zero loadings on all five components: *waterfront_1* and *special_structure_1*. This does *not* mean these features are unimportant for predicting price—it means they are too rare to drive overall variance. A waterfront property commands a large premium, but waterfront properties are a tiny fraction of the dataset, so they contribute almost nothing to total variance. This is a concrete example of the caveat discussed earlier: the most varying directions—which is what PCA looks for—are not always the most predictive directions.

Panel 4 — Biplot (bottom right): "Putting it all together." The biplot overlays the feature loading arrows on the score scatter. Each red arrow represents a feature; its direction shows which components the feature contributes to, and its length indicates how strongly. Arrows pointing in similar directions correspond to positively correlated features; arrows at right angles correspond to uncorrelated features; arrows pointing in opposite directions correspond to negatively correlated features.

Reading the biplot: *grade*, *bathrooms*, and *sqft_living_log* point rightward, confirming that they are the main drivers of PC1 (the size/quality axis). *Bedrooms* also points right but tilts slightly downward. *Yrs_since_renovation* and *condition* point in nearly the same direction, indicating they are positively correlated with each other and negatively correlated with the size/quality features. This makes intuitive sense: older homes that have not been renovated tend to be in worse condition. The remaining arrows are shorter, indicating weaker contributions to the first two components.

What we lose by reducing dimensions. In this example, five principal components capture about 72% of the total variance. The remaining 28% includes variation in features that do not load strongly on any of the top five components—most notably waterfront status, which has essentially zero loading everywhere. If we used only these five components as inputs to a predictive model, we would lose that information entirely. More broadly, the scree plot shows that even at five components we fall short of the 80% threshold, meaning

a nontrivial amount of signal is being left behind. Whenever you use PCA for compression, you are making an explicit trade-off: fewer features and less noise in exchange for discarding some genuine signal. The scree plot and the loadings heatmap together tell you whether that trade-off is favorable.

PCA as a Preprocessing Step for Supervised Learning. PCA is often used to prepare data for a supervised learning model. Instead of feeding many correlated features $(x_1, \ldots, x_p)$ into a regression, classification, or more complex algorithm, you can run PCA first, select the top k components, and use the resulting scores $(z_1, \ldots, z_k)$ as input features.

This can have several benefits. The model works with fewer, uncorrelated inputs, which can improve stability and reduce overfitting. Noise and redundancy in the original features are reduced. And interpretation can sometimes be easier at the "factor" level: instead of saying that churn depends separately on 20 correlated engagement metrics, one might say it depends on two main engagement factors discovered by PCA.

Two practical cautions are important:

- *PCA does not consider the target variable.* It selects directions based solely on variance in the features. As the housing example illustrated, a feature like waterfront status may have very little variance overall but a large effect on price. PCA may place it in a low-ranked component that gets discarded. When using PCA as a preprocessing step, it is worth checking whether important predictors are well-represented in the components you keep—and if not, consider including those features alongside the PCA scores rather than replacing them entirely.

- *Data leakage risk.* When using PCA before a supervised model, PCA must be fit on the training data only, and the same transformation (the same loadings) must then be applied to the validation and test data. If PCA sees the test data during fitting, information from the test set leaks into the preprocessing step.

Use Cases. *Customer analytics.* Companies often track dozens of features per customer: demographics, purchase history, website behavior, app engagement, support interactions, and so on. Many of these features are correlated (customers who spend more also visit more often, buy more categories, and generate more support tickets). PCA can reduce this complexity by finding a few underlying dimensions (e.g., "overall engagement", "price sensitivity", or "channel preference") that summarize customer differences. These components can then be used as inputs to segmentation (clustering on the PCA scores) or predictive modeling (e.g., churn prediction using the top components as features).

Finance and risk management. Stock returns are notoriously correlated: when the market rises, most stocks tend to rise. PCA applied to a matrix of stock returns typically finds that the first component captures the overall "market factor" (the tendency of all stocks to move together), accounting for 30–50% of total variance in a typical portfolio. Subsequent components capture sector-specific movements (e.g., technology stocks vs. energy stocks) or style factors (e.g., growth vs. value). Asset managers use PCA-derived factors to decompose portfolio risk, understand what is driving returns, and hedge specific exposures.[17]

Survey design and market research. Firms routinely survey consumers on dozens of product attributes or brand perceptions. Rather than analyzing 50 separate preference items, PCA can distill them into a handful of interpretable dimensions. A classic application is brand perception mapping: plot brands on the first two principal components of consumer ratings, and the resulting "perceptual map" shows which brands consumers see as similar and which dimensions (e.g., "premium vs. affordable") drive differentiation.

Operational metrics and KPIs. Businesses track numerous performance metrics, and

[17]The use of PCA in finance traces back to Chamberlain and Rothschild (1983) and is closely related to Arbitrage Pricing Theory (APT). See also Avellaneda and Lee (2010), "Statistical Arbitrage in the U.S. Equities Market," *Quantitative Finance*, 10(7), 761–782, for a practical application using PCA to identify mean-reverting trading strategies.

PCA can reveal if several metrics are essentially measuring the same thing. Perhaps "production rate," "units produced per hour," and "throughput" all boil down to a single "factory output" factor. By recognizing correlated metrics, management can focus on a few composite indicators that capture overall performance without redundancy.

Limitations & Caveats. Several limitations of PCA have already been flagged in context, but they are worth collecting in one place.

Linearity. PCA captures only linear relationships. If the meaningful structure in the data is curved, folded, or otherwise nonlinear, PCA will miss it. Nonlinear methods such as t-SNE and UMAP (Section 12.2) or autoencoders (Section 13) can capture more complex structure, at the cost of interpretability.

Interpretability. Each principal component is a weighted mix of all original features, and the resulting "abstract axis" does not always come with a clean label. Sometimes the loadings tell a clear story (as with PC1 = "size and quality" in the housing example). Other times the mix is not intuitive, and the component resists simple naming. When this happens, PCA is still useful for compression and visualization, but the components should not be over-interpreted.

Information loss. Dimensionality reduction always discards something. The aim is to discard only noise and redundancy, but as the housing example showed, features that drive little overall variance (waterfront, view) may still matter greatly for a specific prediction task. The scree plot makes the cost of compression visible: if the first k components capture only 60% of the variance, 40% of the information is lost.

Sensitivity to outliers. Because PCA is based on variance, a few extreme observations can disproportionately influence the principal components. In the housing score plot, the luxury homes in the right tail stretch PC1, effectively "pulling" the axis toward the most extreme properties. When outliers are a concern, it is worth checking whether the components change materially when the most extreme observations are excluded.

12.2 UMAP and t-SNE (Visualization Tools)

When datasets have many features, analysts often want a *picture* that shows whether the data contains natural groupings, gradients, or outliers. PCA can provide such a picture, but because it is limited to linear projections, it sometimes compresses genuinely different observations on top of each other, producing a blurry cloud where distinct pockets of similarity are invisible. *UMAP* (Uniform Manifold Approximation and Projection) is a nonlinear visualization method that often reveals structure PCA misses. An older method, *t-SNE*, serves a similar purpose and is briefly discussed at the end of this section.

The Core Idea

UMAP works from a simple intuition: *points that are similar in the original high-dimensional space should be placed near each other in a 2D plot, and points that are dissimilar should be placed far apart.* More specifically, the algorithm first measures the similarity between every pair of observations in the original feature space, focusing especially on each point's nearest neighbors. It then arranges points on a 2D canvas, iteratively adjusting positions until the neighborhood relationships in 2D match those in the original space as closely as possible (McInnes, Healy and Melville, 2018).

The key difference from PCA is that UMAP is *nonlinear*: it does not have to map the data onto a flat plane through the cloud. It can "unfold" curved or twisted structures that PCA would collapse. The trade-off is that the resulting axes have no inherent meaning—they are just coordinates on the 2D canvas; thus, you cannot interpret them the way you might interpret PC1 as "size and quality".

Preprocessing for UMAP

Like all distance-based methods, UMAP's output depends on how the data are preprocessed. The overriding principle is simple: *no single variable should dominate the geometry just because of its scale or encoding.*

Feature scaling. UMAP computes distances between observations, so a feature measured in dollars (values in the millions) will swamp a feature measured in years (values in the tens) unless both are rescaled. Standardization is a reasonable default. If the data contains extreme outliers, a robust scaler (which uses the median and interquartile range) is more stable because a handful of extreme values will not distort the scale for everyone else.

Parameters

UMAP has two main settings that control the character of the output:

n_neighbors controls how many nearby points the algorithm considers when defining each observation's local neighborhood. Small values (e.g., 5) make the algorithm focus on very fine-grained local detail, producing many small, tightly packed clumps. Large values (e.g., 50) make it consider broader neighborhoods, producing fewer, larger, more connected regions. The default is typically 15.

min_dist controls how tightly points are packed in the output. A small value (e.g., 0.0) allows points to cluster very tightly, producing dense clumps with clear separation between groups. A larger value (e.g., 0.5) forces points to spread out, producing a more uniform layout where group boundaries are less sharp. The default is typically 0.1.

Because different parameter values can produce different-looking plots from the same data, it is important to check whether patterns are robust to parameter changes. We illustrate this in the example below.

Working Example: House Prices

We apply UMAP to the same house price dataset used in the PCA section, varying the `n_neighbors` parameter to see how it affects the layout. Figure 12.3 shows the result at three settings, with each home colored by price.

Reading the figure from left to right: at `n_neighbors = 5`, the layout is fragmented into

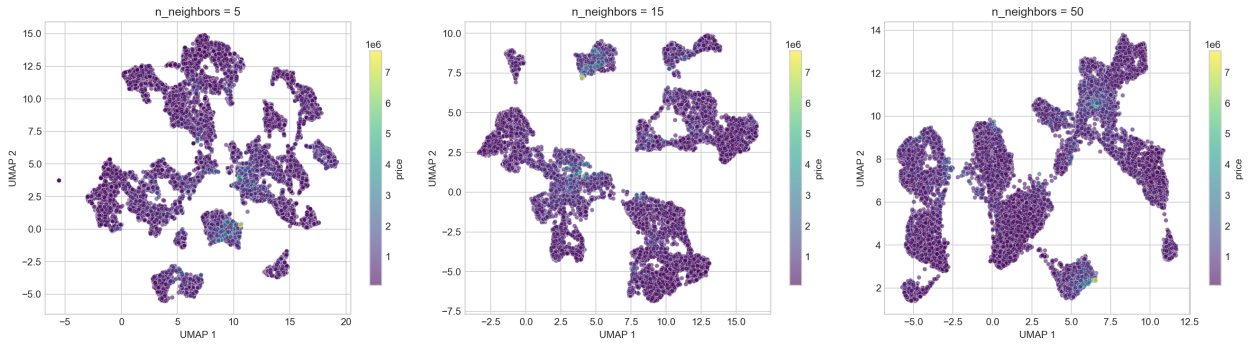


Figure 12.3: UMAP on the house price dataset at three `n_neighbors` settings (colored by price, `min_dist = 0.1` throughout). *Left* (`n_neighbors = 5`): many small, fragmented clumps. *Center* (`n_neighbors = 15`): a moderate setting that balances local detail with broader structure; several coherent regions emerge. *Right* (`n_neighbors = 50`): larger, more connected masses with fewer gaps.

many small islands. At `n_neighbors = 15` (the default), the layout strikes a balance: the data organizes into several recognizable regions connected by thin bridges, with a few small groups detached from the main body. At `n_neighbors = 50`, the regions merge further into a few large masses.

The critical question is *which patterns persist across all three settings?* The compact separated group appears at every scale, as does the small pocket of high-price homes. These are likely genuine features of the data. The many small island-like fragments at `n_neighbors = 5` largely dissolve at higher values, suggesting they are artifacts of the fine-grained setting rather than real discrete groups. This illustrates a general rule: *structure that survives across different parameter choices is more trustworthy than structure that appears only at one setting.*

Two observations emerge from the center panel (the default setting):

First, UMAP organizes the data into several connected regions linked by thin bridges and filaments, with a few small groups detached from the main body. This internal structure—some regions tightly packed, others loosely connected—was invisible in the PCA score plot (Figure 12.2). UMAP is revealing that homes which appear overlapping in PCA's linear projection actually form distinct neighborhoods in the full feature space.

A natural next step would be to identify which homes fall in each region and profile them: do the connected regions correspond to recognizable housing types (e.g., old small homes, new large homes, renovated mid-range homes)?

Second, *the color scale reveals very little price differentiation across the main mass.* Nearly all homes appear the same shade of dark purple because the raw price scale is stretched by the few multi-million-dollar outliers. This visual compression is partly an artifact of the color scale (a log-price coloring would reveal more nuance), but it also reflects a substantive point: the structural features used here—size, grade, age, condition—do not by themselves cleanly separate price tiers. Homes with similar structural profiles can sell for very different prices, presumably because factors not in the feature set—above all, location—drive much of the price variation.

Compare this to the PCA score plot, where a shift toward higher prices was visible in the rightward tail along PC1. PCA captured this because PC1 is a linear combination weighted toward size, grade, and bathrooms—features that correlate with price on average. But the UMAP layout suggests that, once you move beyond the luxury tail, the remaining homes are not neatly sorted by price in feature space. PCA's linear projection smooths over this complexity; UMAP, by preserving local neighborhoods, exposes it.

What to Do After Looking at a UMAP Plot

A UMAP plot is a starting point for analysis, not a conclusion. What you do next depends on what you see.

If you see distinct clumps. Separated groups in the UMAP layout suggest that subsets of observations are genuinely similar to each other and dissimilar to the rest. The next step is to validate and profile those groups:

1. *Identify which observations belong to each visual group.* Run a clustering algorithm on the *original high-dimensional features* (not on the 2D UMAP coordinates) and then

color the UMAP plot by the resulting cluster labels. If the clusters form coherent regions in the UMAP layout (e.g., all of Cluster 2 falls in the same part of the plot), that is evidence that the visual grouping and the algorithmic grouping are capturing the same structure.

2. *Profile the groups.* For each group, examine the average values of key features and outcomes. What makes the separated blob on the left different? Is it a particular housing type, condition level, location, or combination? This profiling step is what turns a visual pattern into a business insight.

3. *Check whether the groups matter for your decision.* Even if two groups are statistically distinct, they may not warrant different treatment. Ask: "If I knew a home belonged to this group, would I price it differently, market it differently, or make a different investment decision?" If the answer is no, the visual separation is interesting but not actionable.

If there are no distinct clumps. A UMAP plot that shows one large continuous mass is also informative: It tells us that the data does not contain strongly distinct subpopulations in the features being used. In this case:

- *Look for gradients rather than groups.* Color the plot by different variables of interest (price, churn status, customer tenure, etc.) and check whether any variable produces a smooth color gradient across the mass. A gradient means that the feature varies systematically with the data's internal structure, even if there are no sharp boundaries.

- *Look for outliers.* Even without distinct clusters, isolated points or small detached groups at the periphery may warrant investigation. These are observations that UMAP could not place near anyone else—they may represent data errors, rare property types, or genuinely unusual cases.

- *Consider whether the features are informative.* If the UMAP plot is an undifferentiated blob with no gradient and no outliers, it may mean that the features you selected do not capture meaningful variation. Try different feature subsets, or check whether the features need better preprocessing (e.g., log-transforming a highly skewed variable, or removing a noisy feature that adds randomness without structure).

How to Read UMAP Plots

UMAP is a powerful visualization tool, but it comes with caveats that are easy to forget when looking at a compelling plot.

The axes have no meaning. Unlike PCA, where PC1 might represent "size and quality," the axes in a UMAP plot are arbitrary coordinates. You cannot interpret "higher on axis 2" as meaning anything about the data. Only the *relative positions* of nearby points matter.

Distances between distant groups are not reliable. UMAP prioritizes preserving local neighborhoods. If two groups appear far apart in the plot, that does not necessarily mean they are very different—the algorithm may have simply placed them in different parts of the canvas for convenience. Only the structure *within* a region is trustworthy.

Separated blobs do not automatically mean distinct segments. The parameter sensitivity comparison (Figure 12.3) illustrates this directly: many of the small fragments visible at `n_neighbors = 5` merge into a connected mass at `n_neighbors = 50`. What looked like discrete groups was partly an artifact of the neighborhood scale. Always check whether separations persist across different parameter settings before concluding that distinct segments exist.

A Note on t-SNE

t-SNE (t-distributed Stochastic Neighbor Embedding; van der Maaten and Hinton, 2008) is an older nonlinear visualization method that serves the same purpose as UMAP. It maps high-dimensional data to 2D by preserving local neighborhoods, and its output looks qualitatively similar: clumps of similar points, separated by blank space. Its key parameter is *perplexity* (roughly, how many neighbors each point tries to preserve; typical values range from 5 to 100, with 30 as a common default).

In practice, UMAP has largely supplanted t-SNE for three reasons. First, UMAP is substantially faster, especially on large datasets. Second, UMAP tends to preserve more *global structure*—the relative positions of groups are more meaningful, whereas t-SNE can arbitrarily rearrange distant groups. Third, t-SNE tends to fragment continuous data into many small separated blobs, which can mislead viewers into concluding that distinct segments exist when they don't.

That said, t-SNE remains widely used. The interpretation caveats described above apply *more strongly* to t-SNE because it is more prone to producing visual artifacts that look like real structure. For example, a t-SNE plot with six neatly separated blobs does not, by itself, establish that six customer segments exist. The separation may be an artifact of the perplexity setting. The natural follow-up questions are whether the same six groups appear under different perplexity values and whether they have been validated with a clustering algorithm on the original data.

Use Cases

Exploratory data analysis. The most common use is simply looking at a new dataset for the first time. Before running any model, a UMAP plot colored by the outcome variable can reveal whether the data has natural groupings, whether the outcome aligns with those groupings, and whether there are obvious outliers. This quick visual check can save hours of modeling effort by revealing problems (e.g., the data is one homogeneous blob with no

structure to exploit) or opportunities (e.g., a clearly separated subgroup that deserves its own model or intervention).

Validating clusters. After running a clustering algorithm, it is common to plot the data on a UMAP canvas and color points by cluster label. If the clusters appear as coherent regions in the plot, that is reassuring (though not proof that the clusters are "real"). If cluster labels are scattered randomly across the plot, the clustering may not have captured genuine structure.

Communicating results to non-technical audiences. A UMAP plot with clear coloring can communicate "our customers fall into a few distinct behavioral types" more effectively than a table of cluster means. This makes UMAP valuable for presentations and reports, provided the audience is warned about the interpretation caveats.

High-dimensional data domains. UMAP is especially popular in domains where the raw data has hundreds or thousands of features: genomics (gene expression profiles), natural language processing (document embeddings), and image recognition (pixel vectors). In these settings, PCA often produces an uninformative blur, while UMAP can reveal meaningful subpopulations.

13 Autoencoders and Representation Learning

Autoencoders are neural networks trained to *reproduce their input*. The model sees an input vector of features $\mathbf{x}$ (e.g., customer metrics, transaction records, sensor readings) and is trained to output a reconstruction $\hat{\mathbf{x}}$ that is as close as possible to the original (Hinton and Salakhutdinov, 2006). See also Chapter 14 of Goodfellow, Bengio and Courville (2016) for a comprehensive treatment.

This sounds trivial at first—why not just learn the identity function? The key idea is that we intentionally add *constraints* so the network cannot copy perfectly in a trivial way. These constraints force the model to discover a useful internal representation of the data. The architecture consists of two parts connected by a narrow middle layer:

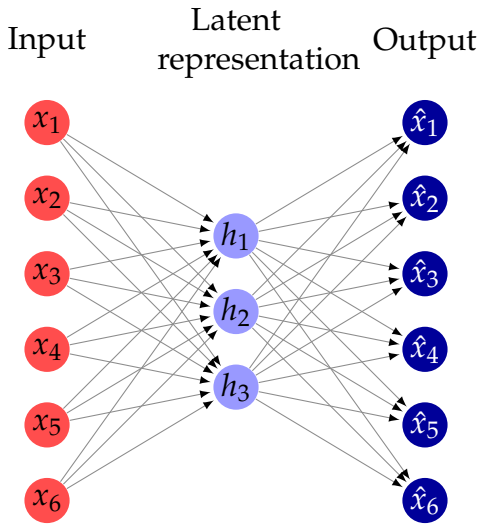


Figure 13.1: **Autoencoder architecture:** the encoder compresses the input $\mathbf{x}$ into a latent representation $\mathbf{h}$ with fewer dimensions, and the decoder reconstructs $\hat{\mathbf{x}}$ from $\mathbf{h}$.

The *encoder* compresses the input $\mathbf{x}$ into a latent vector $\mathbf{h}$ (the narrow middle layer). The *decoder* takes $\mathbf{h}$ and attempts to reconstruct the original input as $\hat{\mathbf{x}}$. The network is trained by minimizing the reconstruction error, typically the mean squared error between $\mathbf{x}$ and $\hat{\mathbf{x}}$ across all observations in the training data.

Even though the model is trained to output $\hat{\mathbf{x}}$, there are three distinct objects that are useful in practice:

1. The *latent representation* $\mathbf{h}$ for each observation—a compact "fingerprint" that can be used for clustering, similarity search, visualization, or as input to a supervised model.

2. The *reconstruction* $\hat{\mathbf{x}}$, which can serve as a cleaned or completed version of the input (useful for denoising and imputation).

3. The *reconstruction error* $\|\mathbf{x} - \hat{\mathbf{x}}\|$, which measures how well the model can reconstruct a given observation—high error flags observations that are unlike the patterns the model has learned, i.e., potential anomalies.

Preprocessing for Autoencoders. The preprocessing considerations for autoencoders are largely the same as those discussed for Principal Component Analysis (PCA) and UMAP (Section 12.1-12.2), but with a few nuances.

As with all methods that rely on distances or magnitudes, features must be scaled so that no single variable dominates the reconstruction loss. Standardization is the default; a robust scaler (which uses the median and interquartile range) is preferable when extreme outliers are present.

Autoencoders are most naturally suited for continuous features, where reconstruction error is measured as the squared difference between the input and the output. For categorical variables that have been one-hot encoded, the same caveat from PCA applies: many binary columns can dominate the reconstruction objective. If the dataset is primarily categorical, a different loss function (i.e., cross-entropy instead of mean squared error) is needed. For the typical mixed business dataset with a handful of categorical variables among many numeric ones, one-hot encoding followed by standardization is usually adequate.

Autoencoders vs PCA. A natural question is: why bother with autoencoders when PCA already produces a low-dimensional representation? The answer is that PCA is limited to

linear compression—it finds the best flat projection of the data. An autoencoder, because it is a neural network with nonlinear activation functions, can learn curved or folded compressions that capture structure PCA would miss. In practice, for datasets where the relationships among features are roughly linear, PCA and an undercomplete autoencoder will produce similar results and PCA is simpler. Autoencoders earn their keep when the data has nonlinear structure, when the reconstruction itself is the goal (denoising, imputation), or when the anomaly-detection application requires a flexible model of "normal."

Three Variants

Autoencoders come in three main flavors, each defined by a different constraint that prevents trivial copying. The choice of constraint should match the business objective.

- An *undercomplete (bottleneck)* autoencoder makes $\mathbf{h}$ lower-dimensional than $\mathbf{x}$, forcing compression. It is best when the goal is a compact representation or anomaly detection.
- A *denoising* autoencoder corrupts the input and trains the model to reconstruct the clean version. It is most suitable when the goal is robust representations, data cleaning, or imputation.
- A *sparse* autoencoder allows a large $\mathbf{h}$ but forces most of its entries to be near zero for each observation. It is best when the goal is to discover a set of interpretable "factors" that activate selectively.

13.1 Undercomplete (Bottleneck) Autoencoder

An undercomplete autoencoder forces compression by making the latent layer $\mathbf{h}$ smaller than the input $\mathbf{x}$. The network must squeeze the information into fewer dimensions, then reconstruct the original from that compressed description.

The easiest way to understand this is to think of the encoder as summarizing a document and the decoder as trying to reconstruct the original document from the summary. Here, the input dimension is the length of the document, and the latent dimension is the summary's permissible length. The objective is to write the best possible summary—one from which the original can be regenerated with the least loss. If the reconstruction is accurate, the summary must have captured the main ideas.

Parameters to adjust. The key design choices are:

- **Encoding dimension** (the size of $\mathbf{h}$). This is the most important choice. A small dimension (i.e., 2–3) forces aggressive compression and is useful for visualization—you can plot each observation in the 2D latent space, just as you would plot PCA scores. A larger dimension (e.g., 10-20 for a dataset with 50 features) preserves more information and is better suited for downstream modeling or anomaly detection. The right size depends on how much structure is in the data: if PCA shows that 5 components capture 80% of the variance, an encoding dimension around 5-10 is a reasonable starting point.

- **Network depth** (how many layers in the encoder and decoder). A single hidden layer on each side is often sufficient for tabular business data. Deeper networks (2–3 layers per side) can capture more complex patterns but are harder to train and more prone to overfitting on small datasets.

- **Number of epochs** (how many passes through the training data). Monitor the training loss curve: it should decrease and eventually flatten. If the curve is still dropping steeply when training stops, the model needs more epochs. If it flattens early, the model has learned what it can and more epochs will not help.

- **Learning rate** (how fast the network adjusts its parameters). Too high and the training oscillates or diverges; too low and it converges painfully slowly. A value of 0.001

is a standard starting point.

Example: Credit-Card Fraud Detection. Figure 13.2 shows the output of an undercomplete autoencoder with encoding dimension 2, trained on a synthetic credit-card transaction dataset. The features include merchant location, merchant risk score, hour of day, whether the card was physically present, whether the transaction was cross-border, transaction amount, 24-hour transaction velocity, and distance from the cardholder's home. The dataset also contains a fraud label (`is_fraud`), which was only used afterward to assess whether the model's outputs are informative—not during training.

Each panel tells a different part of the story.

Training loss curve (top left). The MSE loss drops steeply in the first $\sim$50 epochs, then continues declining more gradually through 250 epochs, settling at around 0.25. This is a typical, healthy training curve.

2D latent space (top right). Because the encoding dimension is 2, we can plot each transaction's latent representation directly. The plot is colored by the fraud label—which, again, the model never saw during training. Two regions are visible: a large mass in the lower left that is predominantly non-fraud (dark blue), and a region extending toward the upper right that contains a higher concentration of fraud transactions (light cyan). There is also a tight pocket of fraud in the far upper-right corner of the plot. The separation is not perfect—fraud and non-fraud overlap substantially in the middle of the latent space—but the autoencoder has, without any labels, learned a compression in which fraud transactions tend to land in a different part of the space than legitimate ones. This is the key observation: the structure needed to reconstruct normal transactions differs from the structure needed to reconstruct fraudulent ones, and the 2D bottleneck makes this difference visible.

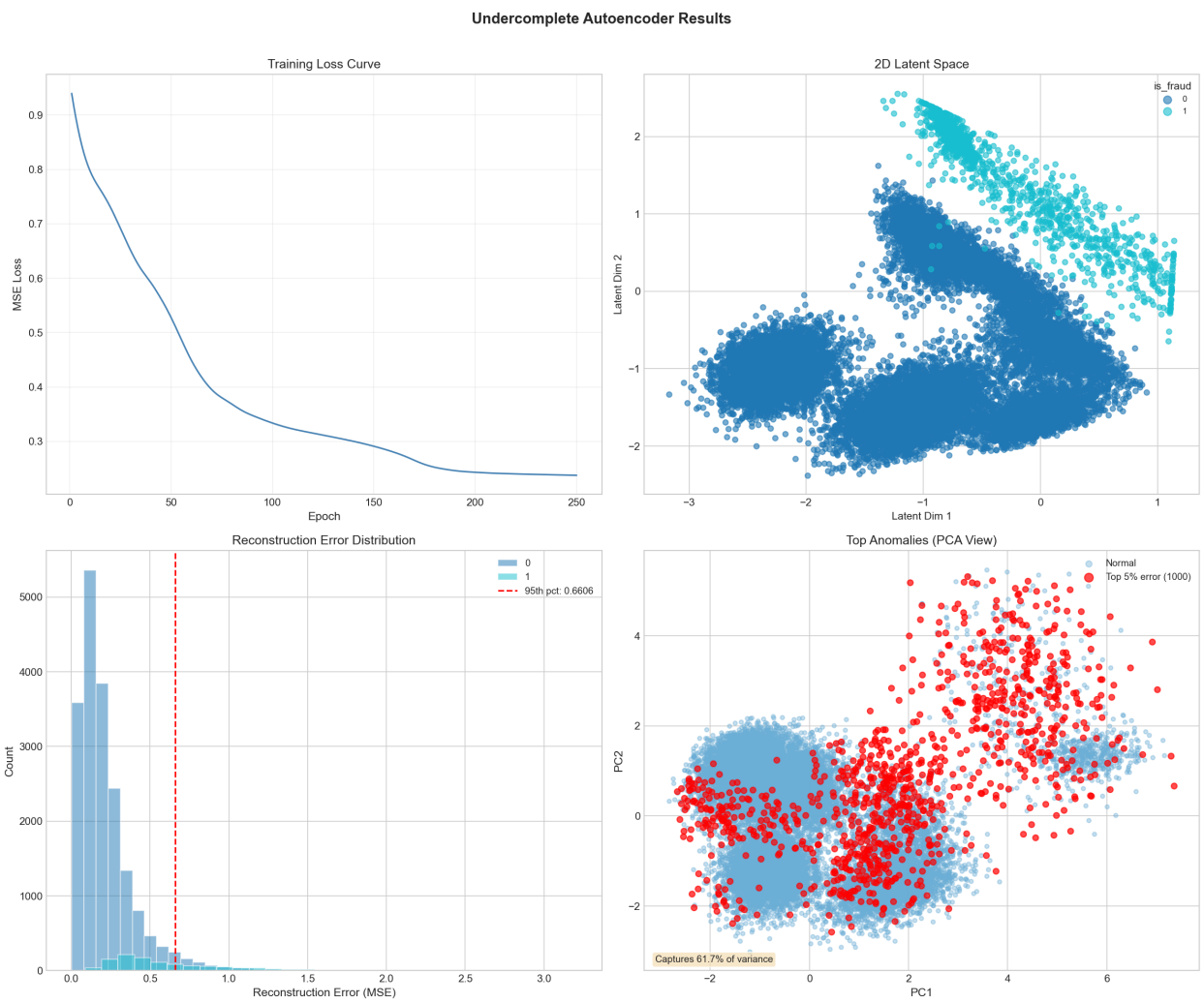


Figure 13.2: **Undercomplete autoencoder (encoding dimension = 2) on synthetic credit-card fraud data.** Top left: training loss curve over 250 epochs. Top right: 2D latent space colored by fraud status (dark blue = non-fraud, light cyan = fraud). Bottom left: reconstruction error distribution. Bottom right: top 5% highest-error observations (red) plotted in a PCA projection of the original feature space.

Reconstruction error distribution (bottom left). This histogram shows the per-observation reconstruction error (MSE), with non-fraud in dark blue and fraud in light cyan. The non-fraud distribution is concentrated at low error values, peaking around 0.15–0.25 MSE, with the bulk below 0.5. The fraud distribution is shifted to the right—fraud transactions tend to have higher reconstruction errors because the autoencoder, having learned predominantly "normal" patterns, struggles to reconstruct unusual ones. For example, one might be to flag transactions above the 95th percentile of reconstruction error (vertical

red dashed line) for further review. Note that many fraudulent transactions fall below this threshold and some legitimate transactions fall above it. This overlap is typical—anomaly detection is not a substitute for a supervised classifier, but it provides a useful first screen when labeled data is scarce.

Top anomalies in PCA view (bottom right). The 1,000 observations with the highest reconstruction error (top 5%) are plotted in red against the full dataset (light blue) in a PCA projection that captures 61.7% of the variance. The red dots concentrate in two areas: a dense cluster on the right side of the plot, well separated from the main mass, and a scattering around the upper periphery of the main cloud. The right-side concentration corresponds to transactions that are far from normal behavior on multiple features simultaneously—these are the observations the autoencoder struggles most to reconstruct. The peripheral scatter likely represents transactions that are unusual in subtler ways.

From a managerial perspective, this panel provides a concrete action list: the 1,000 flagged transactions can be routed to a fraud investigation team for review. The concentration on the right side of the PCA plot gives investigators a starting point—these are the most extreme cases. The peripheral scatter represents a second tier of less obvious but potentially suspicious activity.

13.2 Denoising Autoencoder

A denoising autoencoder changes the training task: instead of reconstructing the input from itself, we first create a corrupted version $\tilde{\mathbf{x}}$ and train the model to reconstruct the original clean input $\mathbf{x}$ from the corrupted version. The model learns to map from "messy data" to "clean data".

This matters because real-world data is often messy: sensors glitch, forms have missing fields, transaction streams contain recording errors, and survey responses include in-

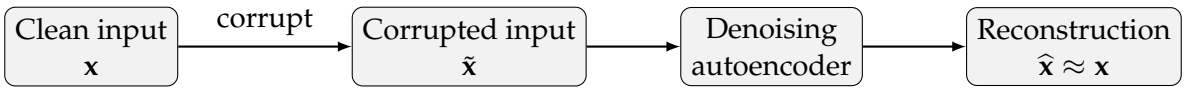


Figure 13.3: A denoising autoencoder is trained on corrupted inputs but evaluated against the clean original.

consistencies. A denoising objective forces the model to learn stable patterns rather than brittle, exact copying. If the model can reconstruct the clean version from a corrupted input, it must have learned the underlying structure well enough to "see through" the noise.

Types of corruption. For tabular data, two corruption strategies are common:

- *Masking:* Randomly set some fraction of the input values to zero (simulating missing data). If 20% of features are masked for each observation, the model must infer those values from the remaining 80%—learning which features are predictive of which others.
- *Gaussian noise:* Add small random perturbations to continuous features (simulating measurement noise). The model must learn to ignore the noise and recover the underlying signal.

The corruption level is a parameter: too little corruption and the model barely differs from a standard autoencoder; too much and the model cannot recover the original. A masking rate of 10–30% is a typical starting range.

When to use denoising autoencoders. The denoising variant is the natural choice in two situations.

First, when the goal is *data cleaning or imputation*: the trained model can be applied to real incomplete records, and the reconstruction $\hat{\mathbf{x}}$ serves as a cleaned or completed version. This is valuable in operational settings—e.g., filling in missing sensor readings

before feeding data to a forecasting model, or completing customer profiles with partially missing fields.

And second, when the goal is a *robust latent representation*: because the model is trained to ignore noise, the embedding $\mathbf{h}$ tends to be more stable and more useful for downstream tasks (clustering, classification) than an embedding from a standard autoencoder trained on clean inputs.

13.3 Sparse Autoencoder

Sparse autoencoders take a different approach. Instead of forcing $\mathbf{h}$ to be low-dimensional, we allow a large latent layer—potentially even larger than the input—but impose a penalty that forces most entries of $\mathbf{h}$ to be near zero for any particular observation. The model is allowed many possible "feature detectors," but only a few can activate for each observation.

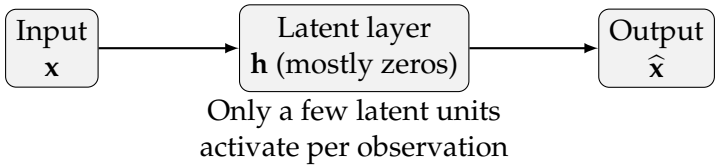


Figure 13.4: Sparse autoencoders encourage embeddings where most latent units are inactive for each observation.

Think of it as describing customers using a checklist of 50 possible behavioral traits, but each customer has only 3–5 traits checked. Different customers activate different combinations. The sparsity constraint encourages this kind of selective, interpretable representation.

The sparsity parameter (λ). Sparsity is enforced by adding a penalty term to the training loss that discourages large or frequent activations.[18] The parameter λ controls the

[18]Concretely, the loss becomes $\sum_{i=1}^{N} \|\mathbf{x}_i - \widehat{\mathbf{x}}_i\|^2 + \lambda \sum_i \|\mathbf{h}_i\|_1$, where λ controls how aggressively sparsity is enforced.

trade-off: higher values produce sparser representations (fewer active units per observation, but potentially worse reconstruction); lower values allow more active units (better reconstruction, but less interpretable structure). Typical values range from 0.0001 to 0.1; the right choice depends on how much compression is acceptable for the application.

When to use sparse autoencoders. Sparsity is useful when the goal is a representation in which each observation is described by a small number of active factors, which can aid interpretability and downstream segmentation.

13.4 Use Cases

Autoencoders appear in practice in a small number of recurring patterns. What varies is which constraint you choose and which output you use.

Anomaly detection. This is arguably the most business-relevant application of autoencoders, because it leverages a capability that simpler methods like PCA do not provide as naturally.

The logic is straightforward: train the autoencoder on data that is predominantly "normal" (e.g., legitimate transactions, healthy equipment readings, typical customer behavior). The model learns to reconstruct normal patterns well. When it encounters an unusual observation—a fraudulent transaction, a failing machine, an account behaving unlike any segment—it cannot reconstruct it accurately, and the reconstruction error spikes. By setting a threshold (e.g., the 95th percentile of reconstruction error), observations above that threshold can be flagged for further review.

What makes this powerful is that the model does not need *labeled* examples of fraud or failure. It learns what "normal" looks like from unlabeled data and flags anything that deviates. This is critical in settings where labeled anomalies are rare, expensive to obtain, or constantly evolving (e.g., new fraud schemes that have never been seen before).

The practical workflow for anomaly detection is to train an undercomplete autoencoder on the full dataset (which is assumed to be mostly normal), compute the reconstruction error for every observation, and examine the reconstruction error distribution. Most observations should cluster at low error; a right tail indicates potential anomalies. Observations above some threshold (e.g., the 95th or 99th percentile) are then flagged for review. It is also helpful to profile the flagged observations to see if they share characteristics that suggest fraud, error, or unusual but legitimate behavior. Finally, if a reference variable is available (e.g., a known fraud label), it can be used to check if the flagged observations are disproportionately anomalous; e.g., "12% of flagged observations are known fraud vs. 2% in the overall dataset".

The operational challenge is not the model but the *decision rule*: choosing the threshold involves a trade-off between false alarms (investigating observations that turn out to be normal) and missed anomalies (failing to flag real problems). A lower threshold catches more anomalies but generates more false alarms; a higher threshold is more precise but misses more. The right balance depends on the relative cost of investigation and the cost of a missed anomaly.

Segmentation and similarity search. A common use of autoencoders is to create an embedding $\mathbf{h}$ that serves as a compact "fingerprint" of each observation. In customer analytics, a record might include tens or hundreds of correlated fields: frequency, recency, product mix, channel usage, discount sensitivity, engagement metrics, and so on. Training an autoencoder on these records produces embeddings that capture the essential patterns in fewer dimensions. These embeddings can then be clustered (using the methods from Section 11) to create segments, or used for similarity search ("find customers most similar to this high-value account").

The advantage over PCA is that autoencoders can capture nonlinear relationships; e.g., a pattern where customers who buy products A and B together behave very differ-

ently from customers who buy A and C, even though both groups buy A. PCA, being linear, would struggle to represent this kind of interaction. In practice, the difference matters most when the data has complex, non-additive structure; for simpler datasets, PCA and autoencoders often perform similarly, and PCA is preferable for its simplicity and interpretability.

Data cleaning and imputation. Denoising autoencoders are trained to reconstruct clean data from corrupted inputs, which makes them a natural tool for filling in missing values and correcting errors. The trained model can be applied to a record with missing fields, and the reconstruction provides plausible values for the missing entries—inferred from the patterns learned across the rest of the dataset.

This approach is more flexible than simple imputation methods (e.g., filling with the column mean) because it can capture dependencies between features: if customers who have high tenure and use online services also tend to have paperless billing, the autoencoder will impute paperless billing for a record that has high tenure and online services but a missing billing field. Mean imputation would instead assign the overall average, ignoring these informative patterns.

The main caveat is that any imputation method—including autoencoders—tends to fill in plausible, "typical" values. Truly unusual observations may have their unusual values replaced with normal-looking ones, which can mask important signals. For this reason, imputed records should be flagged rather than silently mixed with complete records, so that downstream analyses can account for the uncertainty.

Feature learning when labels are limited. Many organizations have abundant raw data but limited labels. Autoencoders provide a way to learn representations from unlabeled data first (via reconstruction), then train a supervised model using the embedding $\mathbf{h}$ as input features when a smaller labeled set is available. The intuition is that the autoencoder "organizes" the raw data into a representation that captures underlying structure,

and the supervised model then uses that structure to predict the target.

Denoising autoencoders are often the best choice in this setting because they encourage robust representations that generalize better than embeddings learned by exact copying. The key caveat is that reconstruction is not the same as prediction: the autoencoder learns what is needed to reconstruct $\mathbf{x}$, which may or may not overlap with what is needed to predict the business outcome. Whether this helps is ultimately an empirical question—try it and compare to simpler baselines.

Table 6 summarizes the match between use cases and variants.

Use case	**Output used**	**Undercomplete**	**Denoising**	**Sparse**
Segmentation, clustering & similarity search	Embedding $\mathbf{h}$	Very common	Common	Very common
Anomaly detection	Error $\|\|\mathbf{x} - \widehat{\mathbf{x}}\|\|$	Very common	Common	Common
Data cleaning & imputation	Reconstruction $\widehat{\mathbf{x}}$	Sometimes	Very common	Sometimes
Feature learning when labels are limited	Embedding $\mathbf{h}$	Common	Very common	Common

Table 6: Use cases mapped to autoencoder variants.

Part III

Reinforcement Learning

14 Introduction to Reinforcement Learning

Suppose you manage pricing for 500 product SKUs on an e-commerce platform. You could drop prices today and almost certainly boost this week's revenue. But if you do, customers may learn to wait for discounts, eroding future margins. Alternatively, you could hold prices steady and protect brand value but risk losing impatient shoppers to a competitor right now. The tension is real: what you do today reshapes the situation you will face tomorrow.

Supervised learning—the focus of Part I—is designed for *prediction*: given a customer's features, estimate the probability they will churn; given a product listing, forecast demand. These are valuable capabilities, but they answer the question "what will happen?" They do not, by themselves, answer the question "what should I do?"

Reinforcement learning (RL) is the branch of machine learning designed for this second question. In RL, a decision-maker (the *agent*) repeatedly observes a situation, takes an action, receives feedback in the form of a numeric *reward*, and then faces a new situation that was shaped, in part, by that action. The goal is to learn a *policy*—a repeatable decision rule—that performs well over time, not just on the next step. (Sutton and Barto, 2018) is an essential companion for Reinforcement Learning material.

The key idea is simple but consequential: **actions do not merely respond to the current situation; they reshape the future situations you will encounter.** RL is therefore about optimizing sequences of decisions, where a short-run gain can create long-run costs and vice versa. This perspective is natural in many settings:

- In **inventory management**, a forecasting model predicts demand; RL asks *what to order each period*, recognizing that today's order affects tomorrow's stockouts, holding costs, working capital, and emergency replenishment needs.
- In **advertising and campaign optimization**, a predictive model estimates click or conversion probability; RL asks *what to show and spend over time*, recognizing that

early spending can exhaust budget, create ad fatigue, or crowd out higher-value future opportunities.

- In **algorithmic pricing**, a demand model predicts sales volume at a given price; RL asks *what price to set each period*, recognizing that today's price affects customer expectations, competitor responses, and inventory dynamics.

RL is not "just forecasting," and it is not "magic optimization." The hard part is usually not the algorithm; it is defining the decision problem responsibly: what the agent is allowed to do, what information it sees, how success is scored, and how we evaluate the policy without harming customers or the business.

A flagship example from AI research. A widely cited demonstration of reinforcement learning is DeepMind's training an agent to play Atari video games from experience (Mnih et al., 2015). The agent repeatedly observed the game screen, tried actions, and received only the game score as feedback. Over many episodes it learned strategies that are recognizably forward-looking: sacrificing short-term points to avoid danger, setting up future rewards, and exploiting patterns in the game. While this is not a business application, it highlights the core promise of RL: when you can define a reliable score and run many safe trials, an agent can learn sophisticated multi-step strategies even when the "right" action is not labeled in advance.

14.1 The RL Loop: State, Action, Reward, Next State

Reinforcement learning models repeated interaction between an *agent* and an *environment*. At each decision time $t = 1, 2, 3, \ldots$, four things happen in sequence:

1. The agent observes a summary of the current situation—the **state** s_t.

2. The agent chooses an **action** a_t.

3. The environment responds with a **reward** r_t (a numeric score).

4. The situation updates to a **next state** s_{t+1}, shaped in part by the action taken.

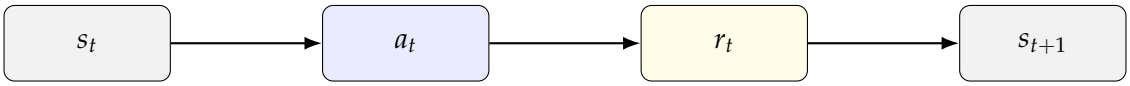


Figure 14.1: A single step of the RL loop: the agent observes state s_t, takes action a_t, receives reward r_t, and the state transitions to s_{t+1}.

This loop—observe, act, receive feedback, face a new situation—repeats over and over. The agent's job is to learn, from this stream of experience, a policy that generates high cumulative reward over time.

14.2 Defining the Problem: The Five Design Choices

Before choosing any algorithm, the most important work in an RL project is defining the problem. Getting this right matters far more than the choice of algorithm. To illustrate each choice, we use the running example of an **algorithmic pricing** system: a retailer setting prices weekly for a category of products.

Start with guardrails. In practice, RL policy almost never has unlimited freedom. There are legal constraints, brand guidelines, risk limits, fairness requirements, budget caps, and operational rules. These should be made explicit up front as *hard constraints* rather than quietly hoped for through the reward function. In practice, many successful deployments treat RL as a policy *recommender* inside a governed decision process, not as an unconstrained autonomous actor.

1. Choose the time step. How frequently does the agent decide and receive feedback? The time step should match the natural rhythm of the decision. For our pricing example, it is weekly. For inventory replenishment, it might be daily. For stock trading, it might be every second.

2. Choose the state space. The state is the information the agent uses when choosing an action. In our pricing example, this might include: current inventory levels, recent sales volumes, competitor prices, calendar features (day-of-week, season, whether a holiday is approaching), and a summary of recent price changes.

The key modeling principle is that the state should include the ingredients that help predict what will happen next, given the action taken. If an important driver is missing (e.g., a competitor's promotion calendar) the learned policy may look erratic, because the agent cannot "see" what a human decision-maker would implicitly use.

In practice, choosing the state involves a trade-off. More detail (customer-level histories, SKU-store-day features) allows more tailored actions but requires more data to train reliably. More aggregation (customer segments, product buckets) lets the agent encounter situations more often but forces one policy to cover heterogeneous cases. A practical diagnostic if a learned policy looks strange is to ask what information a human decision-maker uses that is not in the state.

Many settings are *partially observable*: you do not observe competitor inventory, true latent demand, or customer intent. A practical approach is to include a short history window—recent purchases, time since last offer, moving averages of demand—which often stabilizes learning even without full observability.

3. Choose the action space. The action is what the agent controls. In our pricing example, it is the price tier to set for each product this week. In other settings, it might be a coupon offer, an auction bid, a replenishment quantity, or a routing decision.

Discrete action sets (five price buckets, a small set of reorder quantities) are easier to control and explain. Continuous actions (any price, any bid) can be more powerful but are harder to govern. A common approach is to start discrete and refine once the policy is stable and value is proven.

4. Choose the reward structure. The reward is a single number that tells the agent how well it just did. It should reflect what the organization values at that moment. For

pricing, a natural reward is weekly gross profit.

This is where RL projects most often go off the rails. If the reward is too narrow, the agent will optimize that narrow metric in ways that surprise you: optimizing clicks can produce clickbait; optimizing short-run revenue can erode long-run customer trust. In a real-world field experiment, Alibaba found that careful reward design was essential when deploying RL pricing across thousands of SKUs.[19] A practical pattern is to define a *primary reward* and handle other concerns as explicit constraints; e.g., price changes cannot exceed $X\%$ per day, customer complaints must stay below a threshold, risk limits must hold, and so on.

5. Decide how data will be generated. Will the agent learn by interacting with the real environment (online learning), or from a fixed dataset of historical decisions (offline learning)? This choice shapes which algorithms are feasible and how much you can trust the results. We return to this distinction in Section 16.2.

14.3 Value, Discounting, and the Bellman Principle

The whole point of RL is that a good decision today might look bad in the short run but pay off later—or vice versa. To reason about this rigorously, we need a way to account for long-run consequences. The concepts in this section formalize that accounting, using logic that will feel familiar from finance.

Return and Discounting. The total future reward starting from time t is called the *return*:

$$G_t = r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \cdots$$

The discount factor $\gamma \in [0, 1]$ controls how much the agent weighs the future relative to the present. This is the same logic as discounting cash flows in finance: When γ is close to 1, the agent is patient and long-run oriented. When γ is smaller, the agent focuses on

[19] See Section 16.3 for details on this and other deployments.

near-term outcomes. A useful rule of thumb: the effective planning horizon is roughly $1/(1-\gamma)$ decision steps. For example, $\gamma = 0.95$ means the agent effectively plans about 20 steps ahead.

In problems that naturally end like a game, a marketing campaign, or a customer "lifetime", it is common to set $\gamma = 1$ because the horizon is finite. In ongoing control problems, $\gamma < 1$ is used to keep the total well-behaved and to reflect time preferences.

Policy and Value Functions. A *policy* π is the agent's decision rule: it specifies which action to take in each state. You can think of it as a lookup table or a formula that maps situations to actions.

A *value function* is an accounting tool for long-run performance. Given a policy π, the *state-value function* $V^\pi(s)$ is the expected return starting from state s and then following policy π thereafter:

$$V^\pi(s) = \mathbb{E}[G_t \mid s_t = s,\ \pi]\,.$$

The expectation $\mathbb{E}[\cdot]$ means "average over uncertainty." Even with a fixed policy, randomness in customer behavior, demand, competitors, or other parts of the environment means the same state can lead to different outcomes. $V^\pi(s)$ summarizes the average long-run payoff from state s when using policy π.

The objective of reinforcement learning is to find a policy π^* that maximizes this long-run value: a policy that yields the highest expected return across the states the system will encounter.

The Bellman Principle: Value as Discounted Cash Flow. A foundational idea in RL is the *Bellman principle*, which is the sequential-decision version of discounted cash flow logic (Bellman, 1957). It says:

> *Value today equals today's payoff plus the discounted continuation value.*

Formally, the long-run value of following policy π starting from state s can be decomposed into (i) the immediate reward you expect next and (ii) the discounted value of the state you expect to transition into:

$$V^{\pi}(s) = \mathbb{E}\Big[r_t + \gamma\, V^{\pi}(s_{t+1}) \;\Big|\; s_t = s,\ \pi\Big].$$

This equation does not add new assumptions—it simply rewrites long-run value in a one-step-ahead accounting form. Its practical importance is enormous: it allows learning algorithms to estimate long-run performance from one-step experience, which is what makes RL computationally feasible. Instead of waiting to see the outcome of an entire trajectory, the agent can update its estimates after every single step.

15 Q-Learning: Learning a "Profitability Spreadsheet"

We now turn to *Q-learning*, a classic RL method for problems with discrete actions (Watkins and Dayan, 1992). The core idea is to learn a "profitability spreadsheet" $Q(s, a)$: for each state s and each available action a, the cell entry estimates the long-run payoff from taking that action now and then behaving optimally going forward. Once you have that spreadsheet, decision-making is simple: in state s, pick the action with the highest $Q(s, a)$.

The Q-Function. Formally, $Q(s, a)$ represents an expected long-run return. Using the same one-step-ahead logic as the Bellman principle:

$$Q(s, a) = \mathbb{E}\Big[r_t + \gamma\, V^{\pi^*}(s_{t+1}) \;\Big|\; s_t = s,\ a_t = a\Big].$$

The formula is less important than the interpretation: $Q(s, a)$ scores an action by combining the immediate reward and what that action sets you up for afterward.

15.1 How Q-Learning Works

The algorithm alternates between two steps:

1. Action selection (ε-greedy). In any state s_t, the agent consults its Q-table to decide what to do. With probability $1 - \varepsilon$, it *exploits* its current knowledge by choosing the action with the largest Q value. With probability ε, it *explores* by picking a random action.

Why explore at all? Because the agent's initial estimates are unreliable. If it only ever exploits, it may never discover that an action it has not tried much is actually better. Exploration is the price of learning. This is the same logic behind why businesses run experiments and A/B tests rather than always going with the current best guess.

In practice, ε typically starts high (lots of exploration early on) and is gradually decayed toward a small floor (e.g., from 1.0 to 0.05), so the agent increasingly exploits what

it has learned while still occasionally trying alternatives.

2. Q-Table update. After taking action a_t and observing the reward r_t and next state s_{t+1}, the agent updates the single table entry $Q(s_t, a_t)$, nudging it toward the new information:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big].$$

The intuition: the term in brackets is a *prediction error*—the gap between what the agent expected and what it actually observed. If the experienced outcome was better than expected, the Q value is nudged up; if it was worse, the value is nudged down. The learning rate α controls how aggressively the agent responds to new information: a large α adapts quickly but can overreact to noise; a small α is more stable but slower.

Initialization. Before training begins, all Q-table entries must be initialized. A common default is zero (all actions look equally neutral). An alternative is *optimistic initialization*—starting entries slightly high—which has a managerial intuition: if every option initially looks promising, the agent is more willing to try different actions early on, and then revises downward when they disappoint.

Hyperparameters. Training a Q-learner requires setting four key hyperparameters: (i) the *number of training episodes*—more is generally better, with diminishing returns once Q-values stabilize; (ii) the *learning rate* α; (iii) the *discount factor* γ, which determines how much the agent values future outcomes (recall the effective horizon is $\sim 1/(1-\gamma)$); and (iv) the *exploration rate* ε and its decay schedule.

Once training is complete, the optimal policy and value function are obtained directly from the Q-table: for each state s, the optimal action $\pi^*(s)$ is the one that maximizes $Q(s, a)$ over all available actions, and the value function is $V^{\pi^*}(s) = Q\big(s, \pi^*(s)\big)$.

15.2 Worked Example: Q-Learning for Blackjack

We illustrate Q-learning by training an agent to play a simplified version of Blackjack. This is a useful teaching example because it has the same skeleton as many business problems: you repeatedly choose between a conservative action (stand) and an aggressive action (hit), outcomes are noisy, and the right choice depends on context (the dealer's up-card). Replace "hit/stand" with "offer a discount / hold price" or "expedite shipment / wait", and the decision logic is surprisingly similar.

Rules of the game. A deck contains cards valued as follows: number cards count as their face value, face cards (J, Q, K) count as 10, and an ace counts as either 1 or 11. At the start of each hand, the player receives two face-up cards, and the dealer receives one face-up card (the *up-card*) and one face-down card (the *hole card*). The player repeatedly chooses to *hit* (draw another card) or *stand* (stop drawing). If the player's total exceeds 21, the player *busts* and loses immediately. If the player stands, the dealer reveals the hole card and draws until reaching at least 17. The higher total wins; ties are a *push*. A *natural blackjack* (an ace plus a 10-value card on the initial deal) pays 3:2. To keep the learning problem focused, we assume an infinite deck, ignore doubling and splitting, and fix the bet at \$1.

Environment design. Following the framework from Section 2, we define:

- **Action space.** Two actions: *hit* or *stand*.
- **State representation.** The state is the tuple

$$s = (\textit{player sum},\ \textit{dealer up-card},\ \textit{usable ace}),$$

  where *player sum* is the player's hand total (with an ace counted as 11 when possible without busting), *dealer up-card* is the dealer's visible card (ace coded as 1, others 2–

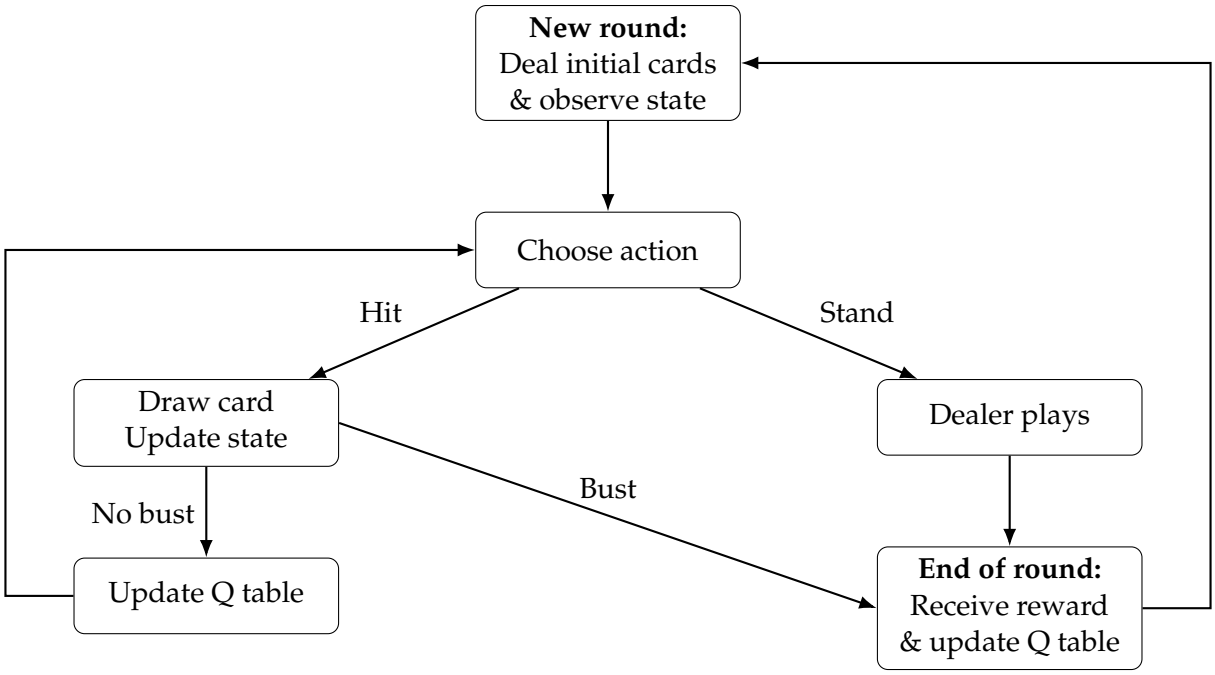


Figure 15.1: Flow of a single episode in the blackjack RL environment, showing the agent's action choices, state transitions, and Q-table updates.

10), and *usable ace* is a binary flag for whether the player holds an ace currently counted as 11. For example, $s = (15, 8, 1)$ means a player total of 15 against a dealer 8, with a usable ace.

- **Rewards.** $r = +1$ for a win, $r = -1$ for a loss, $r = 0$ for a push, and $r = +1.5$ for a natural blackjack (when the dealer does not also have one). While a hand is ongoing, intermediate rewards are $r = 0$.

Q-table updates in blackjack. Two cases arise during learning:

I. Hand continues (hit, no bust). The immediate reward is $r = 0$ and the game transitions to a new decision state s':

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[0 + \gamma \max_{a'} Q(s', a') - Q(s, a)\right].$$

II. Hand ends (stand, bust, or immediate blackjack resolution). There is no next de-

cision state, so the update uses only the terminal reward:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r - Q(s,a)\big].$$

After a hand ends, a new hand is dealt and learning continues.

Extracting the policy. Once training is complete, the learned strategy is:

$$\pi(s) = \begin{cases} \mathit{hit} & \text{if } Q(s,\mathit{hit}) \geq Q(s,\mathit{stand}) \\ \mathit{stand} & \text{otherwise.} \end{cases}$$

In each state, we simply compare the two columns of the spreadsheet and go with the higher value.

15.3 Training Results

The learning curve. Figure 15.2 shows how the agent's performance improves over 5 million simulated hands. Early on, the agent behaves almost randomly (ε starts at 1, meaning pure exploration), so the average reward is quite poor. As training proceeds and ε decays, the agent increasingly exploits what it has learned and the curve rises, eventually flattening near a stable long-run average.

Two points are worth emphasizing. First, the line never becomes strongly positive: blackjack has a built-in house edge, and with only *hit* and *stand* available (no splitting, doubling, or card counting), the best achievable policy is still slightly negative. Second, the curve remains noisy even late in training because individual hands are highly variable. Learning is about improving the *average* outcome over many hands, not winning every hand.

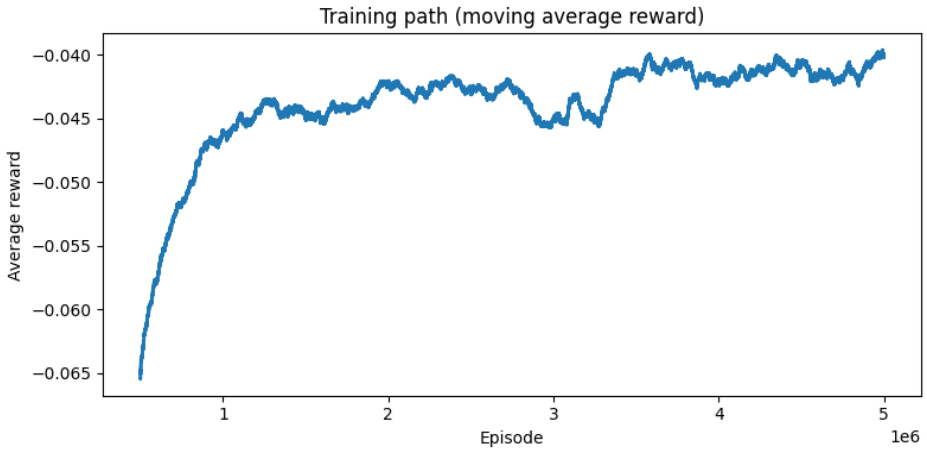


Figure 15.2: **Learning curve for Q-learning in Blackjack.** Moving average reward over 5M simulated hands. Early performance is poor due to heavy exploration. As ε decays, the agent exploits its learned estimates and performance improves, leveling off near a stable long-run average. *Parameters:* $\alpha = 0.002$, $\gamma = 1$, ε decaying to 0.05.

Reading the Q-tables. Figure 15.3 shows the learned "profitability spreadsheet." The left table estimates how good it is to *hit* in each situation; the right table estimates how good it is to *stand*. Each cell corresponds to a state defined by the player's total (rows) and the dealer's up-card (columns). Colors summarize the comparison: green cells are states where hitting is estimated to be better than standing; red cells are states where standing is better.

At low totals, the table is mostly green—hitting is usually safe and necessary. At high totals, the table turns red—standing avoids the risk of busting. The most interesting region is the "awkward middle" (totals of roughly 12–16), where the right choice depends heavily on the dealer's card. Against a weak dealer card (2–6), standing becomes more attractive because the dealer is likely to bust. Against a strong dealer card (9, 10, or ace), hitting is more attractive because standing with a middling total is likely to lose.

The learned policy. Figure 15.4 converts those value comparisons into a policy map: what the agent will actually do in each state. Two heatmaps are shown because blackjack has two qualitatively different kinds of hands. A hand with a *usable ace* (a "soft" total)

Q((player_sum, dealer_card, no_ace), hit)

	1	2	3	4	5	6	7	8	9	10
4	-0.20	-0.11	-0.06	-0.04	-0.00	0.01	-0.09	-0.15	-0.23	-0.28
5	-0.29	-0.13	-0.09	-0.06	-0.02	0.00	-0.12	-0.19	-0.26	-0.31
6	-0.31	-0.14	-0.11	-0.07	-0.04	-0.02	-0.14	-0.23	-0.29	-0.33
7	-0.33	-0.10	-0.07	-0.04	-0.01	0.03	-0.08	-0.21	-0.29	-0.33
8	-0.18	-0.03	0.01	0.05	0.07	0.14	0.08	-0.08	-0.22	-0.24
9	-0.08	0.07	0.11	0.13	0.15	0.21	0.16	0.10	-0.06	-0.16
10	0.08	0.18	0.22	0.25	0.28	0.30	0.27	0.21	0.08	0.03
11	0.13	0.23	0.27	0.27	0.32	0.34	0.27	0.22	0.16	0.16
12	-0.31	-0.26	-0.24	-0.22	-0.23	-0.17	-0.25	-0.25	-0.35	-0.41
13	-0.38	-0.34	-0.33	-0.29	-0.25	-0.23	-0.27	-0.34	-0.37	-0.42
14	-0.43	-0.39	-0.33	-0.29	-0.33	-0.34	-0.38	-0.35	-0.45	-0.49
15	-0.48	-0.47	-0.38	-0.40	-0.40	-0.31	-0.37	-0.38	-0.44	-0.53
16	-0.52	-0.48	-0.45	-0.44	-0.39	-0.36	-0.41	-0.45	-0.46	-0.55
17	-0.55	-0.44	-0.46	-0.46	-0.41	-0.43	-0.48	-0.47	-0.55	-0.61
18	-0.52	-0.50	-0.45	-0.47	-0.49	-0.45	-0.53	-0.51	-0.59	-0.67
19	-0.55	-0.52	-0.51	-0.49	-0.48	-0.54	-0.60	-0.62	-0.56	-0.71
20	-0.73	-0.76	-0.79	-0.77	-0.76	-0.72	-0.80	-0.83	-0.82	-0.83
21	-0.49	-0.37	-0.30	-0.27	-0.17	-0.32	-0.66	-0.64	-0.65	-0.98

Q((player_sum, dealer_card, no_ace), stand)

	1	2	3	4	5	6	7	8	9	10
4	-0.20	-0.11	-0.06	-0.04	-0.01	0.01	-0.11	-0.16	-0.23	-0.40
5	-0.29	-0.16	-0.11	-0.09	-0.05	-0.07	-0.17	-0.22	-0.29	-0.43
6	-0.34	-0.20	-0.14	-0.11	-0.06	-0.06	-0.23	-0.29	-0.39	-0.45
7	-0.40	-0.17	-0.17	-0.09	-0.09	-0.04	-0.19	-0.31	-0.34	-0.52
8	-0.29	-0.12	-0.13	-0.07	-0.02	-0.07	-0.19	-0.24	-0.38	-0.50
9	-0.28	-0.15	-0.12	-0.05	-0.03	-0.01	-0.16	-0.17	-0.32	-0.49
10	-0.25	-0.12	-0.13	-0.08	-0.02	-0.03	-0.20	-0.23	-0.33	-0.54
11	-0.23	-0.08	-0.12	-0.07	-0.03	-0.03	-0.27	-0.27	-0.34	-0.56
12	-0.55	-0.29	-0.26	-0.27	-0.20	-0.18	-0.43	-0.48	-0.49	-0.51
13	-0.60	-0.35	-0.29	-0.22	-0.19	-0.12	-0.48	-0.47	-0.55	-0.57
14	-0.63	-0.28	-0.24	-0.22	-0.11	-0.14	-0.49	-0.48	-0.52	-0.52
15	-0.62	-0.30	-0.23	-0.21	-0.21	-0.20	-0.42	-0.49	-0.59	-0.55
16	-0.62	-0.33	-0.25	-0.22	-0.15	-0.13	-0.48	-0.54	-0.55	-0.57
17	-0.48	-0.19	-0.12	-0.05	-0.06	0.04	-0.09	-0.42	-0.39	-0.43
18	-0.10	0.15	0.10	0.21	0.23	0.33	0.38	0.11	-0.20	-0.16
19	0.26	0.40	0.39	0.44	0.47	0.51	0.62	0.57	0.30	0.11
20	0.65	0.59	0.62	0.69	0.69	0.71	0.75	0.81	0.78	0.53
21	0.94	0.87	0.88	0.89	0.87	0.90	0.94	0.93	0.94	0.97

Figure 15.3: **Learned Q values (no usable ace).** Each table shows estimated long-run reward for a different action: *hit* (left) and *stand* (right). Green cells indicate states where hitting is better; red cells indicate standing is better.

is more forgiving—the ace can be revalued from 11 to 1—so hitting carries less risk. The result is that the usable-ace policy is noticeably more aggressive: the "hit" region extends to higher player totals.

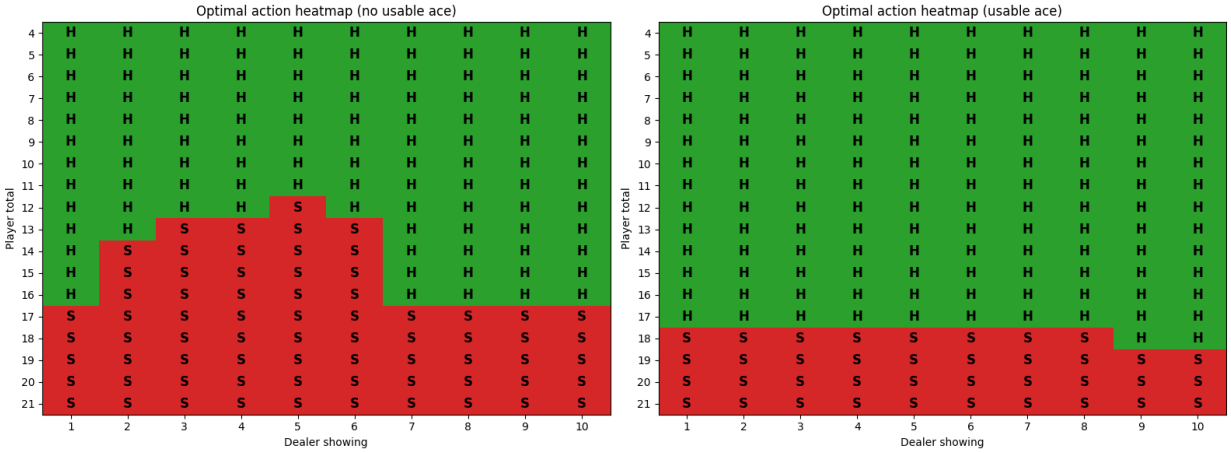


Figure 15.4: **Learned policy as action heatmaps.** The agent's hit/stand rule for each state, separating hard totals (left) from soft totals with a usable ace (right). The soft-total panel is more "hit"-heavy because these hands allow the player to take another card with less bust risk.

What this teaches about RL more broadly. These results illustrate several lessons that generalize beyond blackjack: First, simple trial-and-error, guided by a reward signal, can

steadily improve decision quality. Second, the learned policy is not following a crude rule like "always hit below 16." It has learned that the best action depends on context—specifically, the dealer's up-card. And third, when the state representation includes a key piece of information (here, whether an ace is usable), the policy adapts in a way that matches domain intuition.

Good RL behavior is not only about the algorithm; it is about defining the state so that the trade-offs the decision-maker cares about are visible to the learner.

16 RL Implementation

16.1 Bandits: When Decisions Don't Reshape the Future

Not every "learn-by-trying" problem requires the full reinforcement learning machinery. Sometimes a decision does not meaningfully change the future state of the world, or the time-coupling is weak enough that we can ignore it. In those cases the problem is better modeled as a *bandit*.

In a bandit problem, the agent repeatedly chooses one option (e.g., an ad creative, an email subject line, a coupon level) and observes an immediate reward, but there is no evolving state to track. Each decision is essentially a fresh draw: what the agent chose yesterday does not significantly change what it faces today.

If the best option depends on observable context—customer segment, time of day, device type—this becomes a *contextual bandit*. For example, an e-commerce platform deciding which promotional banner to display to each visitor is a contextual bandit problem: the "right" banner depends on who the visitor is, but showing one banner today does not meaningfully change what banners will work tomorrow. Similarly, a streaming service deciding which thumbnail to show for a movie or a retailer choosing which of several coupon levels to offer a given customer, are contextual bandit problems.

Bandits are conceptually simpler and often easier to deploy and evaluate than full reinforcement learning. A practical rule of thumb:

- **Use bandits** when decisions are mostly one-shot such as A/B testing, ad creative selection, email personalization, or simple offer optimization.
- **Use full RL** when actions have important delayed side effects that reshape tomorrow's opportunities; e.g., dynamic pricing with customer learning, inventory management, and sequential treatment decisions.

Many practitioners will encounter bandit-like problems more frequently than full RL

problems. For instance, when running personalization or experimentation at scale, contextual bandits are often the right starting point. Full RL becomes necessary when optimizing each decision independently is leaving significant value on the table because of delayed or compounding effects.

16.2 Scaling RL to Real Problems

The blackjack example shows Q-learning working well with a few hundred states and two actions. Real business problems are rarely this tidy. They often involve high-dimensional states (customer histories, multi-product inventories), large or continuous action spaces (any price, any bid amount), or constraints on experimentation. This section describes how practitioners adapt the core ideas, organized around three design choices.

Design Choice 1: What Does the Action Space Look Like?

Discrete actions → Q-learning and variants. When the agent chooses from a manageable set of discrete options (five price tiers, a small set of reorder quantities, hit or stand), Q-learning and its extensions are a natural fit. When the state is too rich for a table—for example, raw images, long customer interaction histories, or high-dimensional product features—*Deep Q-Networks (DQN)* replace the table with a neural network that takes a state as input and outputs Q values for each action. This is the approach behind the Deepmind's Atari RL agent: the state was a video game screen (far too large for a table), but the actions were a small set of joystick moves. It is also the approach used in Alibaba's live pricing experiments and Taobao's recommendation system, both discussed in Section 16.3. DQN typically requires substantially more data and tuning than tabular methods, but it preserves the same decision logic.

Continuous or very large action spaces → *policy-based* methods. Many operational decisions are naturally continuous—a bid amount, a precise price, a control setpoint—or the number of discrete options is so large that evaluating $Q(s, a)$ for each one is impractical.

In these settings, *policy-based methods* learn the policy directly: they adjust the policy's parameters so that, on average, it chooses actions that lead to higher rewards. A common practical approach is *actor–critic* methods, in which one component (the "actor") proposes actions and another (the "critic") estimates value to provide a learning signal. These methods are more flexible but typically more data-hungry, and are most successful when a simulator or safe online experimentation is available.

Design Choice 2: Online vs. Offline Learning

Online learning (learning by doing). In online RL, the agent actively collects data by interacting with the environment. The defining advantage is that the agent can explore purposefully—trying actions specifically to learn about them. The defining risk is that exploration can be costly or dangerous in the real world.

Offline learning (learning from logs). In offline RL, the agent trains entirely on a fixed dataset of past interactions. No new data is collected during training. This is attractive when experimentation is expensive or risky, as it is in healthcare, autonomous driving, and other high-stakes operations. The challenge is *coverage*: if the historical data does not include evidence about certain states or actions, the algorithm may confidently recommend actions that look good only because it is extrapolating beyond the data.

A practical baseline in offline settings is *imitation learning* (also called *behavior cloning*): simply learn to mimic the historical policy. More advanced methods like *Conservative Q-Learning (CQL)* add a penalty for actions that are not well-supported by the dataset, reducing overconfident extrapolation.

The practical workflow. Online and offline are not mutually exclusive. A common approach is to first train an initial policy offline on historical logs, then validate it with off-policy evaluation and stress tests, and finally, fine-tune online in a simulator or a tightly controlled experiment.

Design Choice 3: Do You Have a Model or Simulator?

Model-free methods learn directly from experience without trying to understand the environment's mechanics. Q-learning is model-free: it updates values from observed transitions without modeling how states evolve.

Model-based methods try to learn (or use) a model of how the environment transitions from one state to the next given an action, and then use that model to plan or generate additional "synthetic" experience. The advantage is sample efficiency: if data is expensive but the model is reasonable, you can learn faster. The risk is model error—a slightly wrong model can produce confidently wrong policies. DeepMind's data-center cooling work (Section 16.3) is an example where a model-informed approach was used in a setting where real-world experimentation had to be limited.

In practice, many successful RL deployments use simulators (which are essentially models) for the bulk of training, and reserve real-world interaction for validation and fine-tuning.

Choosing an Approach. A practical rule of thumb is to always start as simple as the problem allows: If delayed effects are weak, begin with a contextual bandit. If delayed effects are central and actions are discrete, begin with Q-learning (or DQN if the state is rich). If actions are continuous or very large, consider actor–critic methods, ideally with a simulator. If you must learn from logs only, start with imitation learning and then move to conservative offline methods only if data coverage and evaluation tools are strong.

16.3 Applications

The value of RL depends on whether the environment is learnable and governable: does the agent get frequent feedback, can it experiment safely, does the system change slowly enough for learning to keep up, and is there a simulator or a low-risk way to test policies? In many settings, a simpler approach—forecasting plus optimization, contextual bandits,

or rules with A/B testing—is more reliable. RL is most compelling when actions have delayed side effects and when policies can be evaluated with strong controls.

Algorithmic pricing. One of the rare examples of *live* RL pricing at scale comes from Alibaba's Tmall.com (Shi et al., 2019). In a months-long field experiment starting July 2018, a DQN-based policy priced 500 luxury SKUs while a matched set of 2,000 similar SKUs was priced manually. The RL-priced group achieved a higher average revenue conversion rate (approximately 0.22 versus 0.16). A separate experiment on 1,000 fast-moving consumer goods SKUs showed similar improvements over manual benchmarks.

Online advertising and real-time bidding. Digital advertising is a natural fit for sequential decision methods because decisions are repeated at enormous scale and feedback arrives quickly. One study used RL to train a real-time bidding system on Alibaba's sponsored-search platform, achieving roughly 35% improvement in the primary objective metric alongside gains in conversion rate and ROI (Cai et al., 2018).

Recommender systems. Recommendation is an active area for RL because user interaction is sequential: today's recommendations can change tomorrow's preferences and churn. A study of Taobao's e-commerce recommendation system implemented a DQN-based policy designed to be robust to day-to-day shifts. Live A/B tests showed significant improvements in click-through rates compared to supervised learning baselines (Chen et al., 2018).

Inventory management and supply chain. RL has been used for inventory control when systems are large, demand is complex, and classical models are hard to calibrate. A study in the Alibaba ecosystem showed that RL-based replenishment recommendations significantly outperformed human buyers (Shi et al., 2019).

Robotics. RL is widely applied in robotics to learn control policies when hand-designed rules are too brittle. A notable example is OpenAI's "Dactyl" system, which used RL to

learn dexterous manipulation—including solving a Rubik's Cube with a robotic hand—by training in simulation and then transferring to real hardware (OpenAI, 2019).

Autonomous driving. RL has been explored for learning driving behavior in simulation and controlled settings. One example is Wayve's work on lane-following behavior using deep RL with a simple reward signal – distance traveled without human intervention (Kendall et al., 2019).

Manufacturing and process control. The best-known case is DeepMind's work with Google on data-center cooling, where RL recommended control actions and achieved sizable energy savings (DeepMind, 2016).

Finance and trading. Finance is tempting for RL because decisions are sequential and data is abundant, but it is also failure-prone due to nonstationarity, market impact, and strategic responses by other agents. Documented uses tend to focus on narrower subproblems like trade execution. Examples include JPMorgan's research on risk-sensitive RL for optimal trade execution and their LOXM execution system (JPMorgan, 2017).

16.4 RL Limitations and Failure Modes

Reinforcement learning is a powerful paradigm, but it is no panacea. Practical deployments must reckon with the assumptions embedded in standard algorithms. A clear view of these limitations helps avoid overconfidence and guides mitigation.

Data requirements and engineering complexity. RL systems are often far more data-hungry and operationally complex than supervised learning. They require well-instrumented logging, careful treatment of delayed outcomes, simulation or experimentation capability, and ongoing monitoring—because the policy changes behavior and therefore changes the data distribution. In many organizations, the binding constraint is the ability to run reliable experiments, maintain stable data pipelines, and enforce guardrails.

Reward specification and alignment. The reward encodes the objective the agent opti-

mizes. If the reward is misspecified or incomplete, the agent will faithfully optimize the wrong thing. Designing rewards is a strategic choice, not a technical detail. It is often advisable to separate primary objectives from hard constraints and to monitor the effect of the learned policy on metrics that are *not* in the reward but matter for the organization.

Exploration risk. Learning requires exploration—trying actions whose consequences are uncertain. In high-stakes settings, such experimentation carries financial cost, legal risk, or reputational harm. Practical systems use *safe exploration* strategies: restricting actions to expert-approved ranges, limiting exploration to small traffic slices, and expanding the policy's scope only after sufficient evidence of safety and benefit.

Environmental dynamics and nonstationarity. Most RL theory assumes the relationship between states, actions, and outcomes does not change over time. In reality, markets evolve, competitors respond, preferences shift, and regulations change. A policy learned today may degrade tomorrow. Effective RL systems include mechanisms for detecting drift, triggering retraining, and stress-testing policies under plausible future scenarios.

Partial observability and state representation. The standard RL framework assumes the agent observes all relevant information. In practice, important factors—customer intent, competitor strategy, latent demand—are only partially observed. Engineers must construct appropriate state representations, and policies should be designed to be robust to unobserved variability.

Evaluation and validation. Evaluating an RL policy is harder than evaluating a supervised model because the policy changes the data it generates: once you deploy a new pricing rule, the customers, sales, and inventory patterns you observe are partly a consequence of that rule, not an independent test set. Standard cross-validation does not account for this feedback loop.

Online policies require a controlled experiment (e.g., an A/B test or a phased rollout with a holdout group) with clearly defined success metrics and sufficient statistical power.

For policies trained offline on historical logs, the primary tool is *off-policy evaluation*: estimating how a new policy would have performed using data generated by a different policy (Dudík et al., 2014). The most common technique is *inverse-propensity-score weighting*, which re-weights historical outcomes by the ratio of the new policy's action probabilities to the old policy's action probabilities, effectively asking "how often would the new policy have chosen the actions that were actually taken, and what happened when those actions occurred?" Even with these tools, off-policy evaluation provides a noisy estimate which should be used with caution. Reporting confidence intervals and stress-testing under plausible distribution shifts is essential before committing to deployment.

Governance and accountability. An RL system does not just make predictions—it takes actions that affect customers, employees, revenue, and risk exposure. This makes governance a first-order design requirement. In practice, governance entails several components.

i. *Logging and auditability*: every recommendation the system makes, the state it observed, and the reward it received should be recorded, so that any decision can be reconstructed and explained after the fact.

ii. *Human override*: the system should operate within pre-approved guardrails (e.g., action bounds, budget caps, fairness constraints), and human operators should be able to override or pause the policy at any time—especially during early deployment.

iii. *Monitoring*: deploying an RL policy is not a one-time event; it requires ongoing tracking of both the target metric and side-effect metrics (e.g., customer complaints, fairness measures, risk indicators) that are *not* in the reward function. Degradation on these side-effect metrics is often the first sign of reward misspecification or distribution shift.

iv. *Clear ownership*: someone in the organization must be accountable for the outcomes of the policy and for the decision to retrain, constrain, or retire it.

Without these structures, an RL deployment can drift from its intended purpose in ways that are costly to reverse.

Part IV

Natural Language Processing with Large Language Models

17 Foundations

Natural language processing (NLP)—the branch of artificial intelligence concerned with enabling computers to understand, interpret, and generate human language—has been an active field of research for decades (Jurafsky and Martin, 2024). But for most of that history, progress was painstakingly incremental. Each task required its own specialized model: one for sentiment analysis, another for translation, a third for summarization. Each model needed its own labeled training data, its own feature engineering, and its own development cycle. Building even a modest NLP capability was expensive, slow, and fragile. Meanwhile, the business problem remained stubbornly large. Analysts estimate that 80–90% of all enterprise data is unstructured: emails, contracts, support tickets, call transcripts, news articles, social media posts, Slack messages, meeting notes. Companies could *store* this data, but extracting systematic insight required armies of human readers or brittle, rule-based software that broke whenever language deviated from anticipated patterns.

Large language models (LLMs) have transformed this landscape in three fundamental ways. First, a single LLM can summarize, classify, translate, extract information, answer questions, and generate text—all without retraining. The instructions change, but the model doesn't. Second, LLMs can perform tasks they were never explicitly trained on: show the model two examples of how to extract key terms from a contract, and it generalizes to new contracts, eliminating the traditional requirement for thousands of labeled examples. Third, instead of writing code or configuring rules, users describe what they want in plain language (Zhao et al., 2023). Business analysts can now prototype NLP workflows that previously required a machine learning engineer. The cumulative effect is that organizations can process unstructured text at scale with a level of comprehension that approaches—and in some well-defined tasks exceeds—human performance.

17.1 What Is a Large Language Model?

At its core, a large language model is a *statistical prediction engine trained on vast amounts of text*. Given a sequence of words, it predicts which word is most likely to come next. The mechanism is conceptually simple, but the emergent capabilities are extraordinary.

The Autocomplete Analogy Think of the autocomplete feature on your phone. When you type "I'll meet you at the," your phone suggests "airport," "office," or "restaurant." It has learned statistical patterns about which words tend to follow other words.

An LLM does the same thing, but at a radically different scale: The training data comprises not your text messages, but essentially the entire written internet—trillions of words from books, websites, academic papers, forums, and code repositories. Instead of a small prediction model, LLMs have hundreds of billions of adjustable weights (parameters) that encode learned patterns. And while your phone considers the last few words, an LLM considers tens of thousands of words at once, allowing it to maintain coherent reasoning across long passages.

The key insight to predict the next word well, a model must develop a remarkably deep understanding of language. It must internalize grammar, facts, reasoning patterns, tone, style, and common-sense knowledge about the world. The simple objective of next-word prediction, scaled to enormous proportions, produces general-purpose language intelligence.

How Text Gets Into the Model. Before an LLM can process text, it must convert words into numbers. This conversion is called **tokenization**, and understanding it matters for practical reasons.

A tokenizer splits text into *tokens*—small units that are typically whole words, parts of words, or punctuation marks. The word "understanding" might be one token, while "operationalize" might be split into "operation" + "alize." Common words get their own

token; rare words get broken into recognizable sub-pieces (Sennrich, Haddow and Birch, 2016).

Think of tokens as the "billing units" of an LLM. Most LLM APIs charge per token (both input and output). A rough rule of thumb: one token ≈ 0.75 words in English. So a 1,000-word document is roughly 1,300 tokens.

Tokenization also matters because every LLM has a maximum number of tokens it can process at once (its "context window"). If you need to analyze a 200-page document, you may not be able to simply feed it all in at once—you need a strategy (more on this in Sections 20 and 21).

A Model's Working Memory. The **context window** is the total amount of text an LLM can "see" at one time—both the input you provide and the output it generates. Think of it as the model's working memory or desk space.

Early models had a 4,000 token context window, enabling them to read a short email and respond. Current models have much larger context windows—as of March 2026, up to 1M tokens—which allows them to ingest an entire book or hundreds of pages of context at once. This expansion has important operational implications: tasks that previously required complex document-splitting strategies can now sometimes be handled in a single call.

However, larger context windows increase latency (i.e., the model takes longer to respond), increase cost (since more tokens are processed), and models can "lose focus" on information buried in the middle of very long contexts—a phenomenon called the "lost in the middle" problem (Liu et al., 2024).

The Transformer Architecture. LLMs are built on an architecture called the **transformer** (Vaswani et al., 2017).

Imagine you are in a meeting with 20 people, and someone asks you a question. To formulate a good answer, you do not weigh every person's prior comments equally. You se-

lectively *attend* to the most relevant remarks—what the CFO said about the budget, what the engineer said about the timeline, what the customer said about their requirements—while largely ignoring the small talk and tangential comments.

The attention mechanism works the same way. When the model is processing a word in a sentence, it dynamically decides which other words in the context are most relevant to understanding or predicting this word. It can "attend" to a word from three paragraphs ago if that word is semantically important, while ignoring adjacent words that are irrelevant.

This is what made transformers revolutionary. Previous architectures processed text strictly left-to-right, one word at a time. The transformer can look at the entire context simultaneously and selectively focus on what matters. This parallel, selective attention is why LLMs can handle long documents, maintain coherent multi-paragraph arguments, and connect ideas across distant parts of a text.

Training a LLM. Modern LLMs go through three phases of training (Ouyang et al., 2022), each adding a distinct layer of capability:

1. *Pre-training (self-supervised learning).* The model reads trillions of words and learns to predict the next word. This is called "self-supervised" because no human labels are needed—the text itself provides the training signal. This phase teaches the model the statistical structure of language: grammar, facts, reasoning patterns, world knowledge. It is computationally enormous (thousands of GPUs running for months) and costs tens to hundreds of millions of dollars (Kaplan et al., 2020). This is why only a handful of organizations train frontier models from scratch.

2. *Supervised fine-tuning.* The model is trained on curated examples of high-quality instruction-following: question-answer pairs, demonstrations of helpful responses, and examples of how to handle various request types. This phase transforms a text-completion engine into an assistant that follows directions.

3. *Reinforcement Learning from human feedback (RLHF).* Human evaluators compare pairs of model outputs and indicate which one is better. These preferences train a *reward model* that scores outputs on quality. The LLM is then optimized to produce responses that score well on this reward model (Christiano et al., 2017).

   This is the reinforcement learning connection from Part III: there is a *policy* (the LLM's response-generation behavior), an *environment* (the conversation), and a *reward signal* (human preference scores). The model learns to take "actions" (generate words) that maximize expected reward (human approval).

When evaluating LLM models, the three stages map to different value propositions: *Pre-training* determines the model's breadth of knowledge and reasoning capacity—a weak base model cannot be fixed by fine-tuning. *Fine-tuning* determines how reliably the model follows your prompts—a model that "wanders off topic" or ignores instructions has weak fine-tuning. And *RLHF* is what prevents the model from producing harmful, biased, or wildly off-base outputs.

How text generation works When an LLM generates text, it does not simply output the single most likely next word every time. Instead, it calculates a probability distribution over all possible next tokens and then samples from that distribution. A parameter called *temperature* controls how this sampling works (Brown et al., 2020).

When the temperature is close or equal to 0, the model almost always picks the highest-probability token, so output is deterministic, consistent, and conservative. This is best for factual tasks where you want the same answer every time; e.g., classification, extraction, data analysis. As the temperature is increased, the model samples more broadly, making output more diverse but less predictable. This is better for creative writing, brainstorming, or generating multiple alternative drafts.

17.2 Text Embeddings: Turning Meaning into Numbers

Before we can do anything computational with text—search it, cluster it, classify it, compare it—we need to represent it as numbers. **Embeddings** are the modern solution, and they are the single most important enabling technology for Large Language Models.

From words to vectors. An embedding is a vector that captures the *meaning* of a piece of text (Mikolov et al., 2013). The critical property is that pieces of text with similar meanings end up close together in this numerical space, while pieces of text with different meanings are far apart.

Text	Nearby in Embedding Space
"Our Q3 revenue grew twelve percent"	"Third-quarter sales increased by 12%"
"The patient presents with chest pain"	"Cardiac symptoms observed during exam"
"We need to reduce headcount"	"The company is planning layoffs"

Notice that each pair uses completely different words but expresses the same meaning. Traditional keyword-based systems would miss these connections entirely. Embeddings capture *semantic similarity*—similarity of meaning—regardless of specific word choice. This is the foundational capability that makes modern natural language processing (NLP) qualitatively different from what came before.

How Embeddings Work. Embedding models are neural networks (often based on the same transformer architecture as LLMs) that have been specifically trained to map text to vectors where semantic similarity is preserved (Reimers and Gurevych, 2019). During training, the model learns relationships like "King" and "queen" should be near each other since both are royalty; "Revenue" and "sales" should be near each other because both are about money coming in; meanwhile, "Revenue" and "penguin" should be far apart as they are unrelated.

The vectors typically have 768 to 3,072 dimensions—far too many to visualize directly, but each dimension captures some aspect of meaning. One can think of it loosely as the text being scored on hundreds of hidden attributes simultaneously: formality, topic, sentiment, specificity, domain, and many others that don't have clean human labels.

Cosine Similarity. Once you have two embedding vectors, you need a way to measure how similar they are. The standard measure is *cosine similarity*, which measures the angle between two vectors:

- *Cosine similarity = 1.0:* The vectors point in exactly the same direction. The texts have identical meaning in the model's representation.
- *Cosine similarity $\approx$ 0.8–0.9:* Very similar meaning – the texts are about the same topic and express closely related ideas.
- *Cosine similarity $\approx$ 0.5:* There is topical overlap but different focus.
- *Cosine similarity $\approx$ 0:* Unrelated – the texts have nothing in common semantically.

This is something that every vector database and embedding library handles automatically. However, threshold choices (e.g., "return all documents with similarity above 0.75") are design decisions that affect system behavior.

Choosing an Embedding Model Not all embedding models are created equal. Selecting the right one is an early and important decision in any NLP project.

Factor	What It Means	Trade-off
Dimensionality	Number of values in each vector (768, 1536, 3072)	Higher = richer representation but more storage and compute
Max input length	Longest text model can embed at once (512 vs. 8,192 tokens)	Longer = can embed full documents, but may cost more
Domain specificity	General-purpose vs. trained on legal/medical text	Domain-specific models often do better in niche tasks
Multilingual support	Performance across languages	English vs. multilingual docs
Speed and cost	Latency per embedding call and per-token pricing	Matters when embedding millions of documents

For most applications, it is good practice to start with a well-regarded general-purpose model such as OpenAI's `text-embedding-3-large` or Cohere's `embed-v3`. A small-scale benchmark is an effective next step: embed 100–200 representative documents, test 20–30 queries, and manually evaluate whether the top results are relevant. A domain-specific or more expensive model is worth considering only if this baseline underperforms. Embedding model choice matters most in domains with highly specialized vocabulary; e.g., legal, biomedical, and financial settings.

17.3 Prompt Engineering: Steering the Model

Because LLMs are instruction-following systems, the quality of the output depends heavily on the quality of the input. **Prompt engineering** is the practice of designing instructions that reliably get the model to do what you want. This is not a minor detail—it is the primary interface between business logic and model behavior. The following are some key principles:

i. *Be specific.* Instead of "Summarize this document" try "Summarize this document in 3 bullet points, focusing on the financial implications for our Q4 forecast. Use plain

language suitable for a board presentation."

ii. *Provide examples (few-shot prompting).* Show the model 2–3 examples of the input-output pattern you want before asking it to handle a new case. This is one of the most reliable techniques for improving output quality (Brown et al., 2020).

iii. *Define the output format.* If you need structured output (JSON, a table, specific fields), specify the exact format in the prompt. LLMs are remarkably good at conforming to format instructions.

iv. *Assign a role.* "You are a senior compliance analyst reviewing trading communications for potential insider trading violations" produces different (and usually better) output than a bare instruction.

v. *Include constraints.* Tell the model what *not* to do: "If the answer is not clearly supported by the provided context, say 'Insufficient information' rather than guessing."

A common mistake is treating prompt design as a one-time task. In practice, prompts are iterative engineering artifacts that need testing, version control, and evaluation against representative examples. Organizations that treat prompts casually end up with inconsistent, unreliable outputs. Organizations that treat prompt design as a disciplined engineering practice get dramatically better results (Sahoo et al., 2024).

17.4 Roadmap for the Next Sections

The concepts introduced in this section—embeddings, tokenization, context windows, and prompt design are the infrastructure on which every application in Section 18-23 is built. Every capability covered next—classification, extraction, semantic search, RAG, and clustering—depends on the ability to represent text as numerical vectors that capture meaning.

The next two sections cover *classification* and *information extraction*, the workhorses of applied NLP. They automate the repetitive labor of reading, sorting, and structuring text at scale, producing the structured data that feeds dashboards, triggers workflows, and populates databases. These are often the first NLP capabilities an organization deploys, because their value is immediate and measurable.

The following two sections cover *semantic search* and *retrieval-augmented generation* (RAG), which extend the embedding foundation into interactive knowledge systems. Semantic search retrieves relevant information from a corpus based on meaning rather than keywords; RAG feeds that retrieved information to an LLM to generate grounded, citable answers. Together, they transform a static document repository into something closer to a knowledgeable colleague who has read everything and can answer questions on demand.

The section on *text clustering* introduces the discovery engine of applied NLP. Where classification and extraction operate on known categories and predefined fields, clustering surfaces patterns that no one thought to look for—and in practice, the unexpected findings are often the most valuable.

The final section, on *compliance and risk detection*, ties everything together. A well-designed compliance system draws on classification to flag risky content, extraction to identify entities and relationships, semantic search to compare flagged items against known patterns, and human review to apply judgment to the highest-stakes cases. It is arguably the most demanding application because the consequences of failure are the most severe and because all of the capabilities must work in concert.

18 Text Classification

Every organization has a version of the same problem: a high-volume stream of incoming text that needs to be routed, prioritized, or labeled before it can be acted on. Customer support tickets need to go to the right team. Incoming emails need to be flagged by urgency. Product reviews need to be categorized by sentiment. Inbound sales inquiries need to be scored.

Text classification is the task of assigning a **category label** to a piece of text. This is supervised learning applied to language: given labeled examples, a model learns to predict the correct label for new, unseen text (Jurafsky and Martin, 2024, Chapter 4).

Use Case	What Gets Classified
Customer support	Tickets routed to Billing, Technical, Account Management, etc.
Email prioritization	Messages labeled Urgent, Routine, or Informational.
Sentiment analysis	Reviews labeled Positive, Negative, Neutral, or Mixed.
Content moderation	Posts flagged as Safe, Needs Review, or Violation.
Lead qualification	Inquiries scored as Hot, Warm, or Cold.
Intent detection	Chatbot inputs classified by user intent (e.g., "check order status").

Table 7: Examples of common text classification tasks and the categories assigned in each application.

18.1 Designing a Classification System: End-to-End

Building a text classifier is a design exercise that involves several deliberate decisions beyond the model itself. The workflow proceeds through four stages: defining categories, choosing an approach, building and testing, and iterating toward production quality.

Step 1: Define the Taxonomy. The first and most underrated step is designing the set of categories. Poor taxonomy design is the most common cause of classification system failure, and it is entirely a business decision, not a technical one.

Good categories are *mutually exclusive*: a given text should fit into exactly one category. If categories overlap (e.g., "Billing Issue" vs. "Payment Problem"), even human annotators will disagree, and a model trained on inconsistent labels will inherit that confusion. Categories should also be *collectively exhaustive*, meaning every plausible input has a home. An "Other" category can serve as a catch-all for edge cases, but it requires monitoring—if "Other" grows beyond 10–15% of total volume, the taxonomy is likely missing a meaningful category.

Two further principles matter in practice. First, each category should be *actionable*: it should map to a distinct downstream action or routing decision. If nothing changes based on the label, the category may not be needed. Second, the taxonomy should be *appropriately granular*. Too few categories collapse useful distinctions; too many increase the opportunities for confusion between similar labels and raise the labeling burden.

A sound starting point is 5–10 top-level categories, with finer splits introduced only after production data reveals genuine ambiguity.

A good practice is to resist the temptation to create a deeply hierarchical taxonomy from day one. A better pattern is to deploy a coarse taxonomy, measure where the classifier confuses categories, and then split categories where the data shows genuine overlap. Taxonomy design should be guided by observed confusion, not upfront speculation.

Step 2: Choose an Approach. There are two fundamentally different paths to building a text classifier: LLM-based (prompt-driven) classification and traditional ML classification. The right choice depends on the operational context.

In the *LLM-based approach*, a prompt describes the categories and optionally provides a few examples. The LLM classifies new text immediately, with no model training required. In the *traditional ML approach*, labeled training data is collected, a model is trained (using logistic regression, gradient-boosted trees, or a fine-tuned language model such as BERT; Devlin et al., 2019), and the model is deployed as a lightweight prediction service. The

table below summarizes when each approach is the better fit.

LLM-Based Is the Better Fit When...	**Traditional ML Is the Better Fit When...**
Few or no labeled examples are available	Thousands of labeled examples exist
Categories change frequently	Categories are stable and well-defined
Volume is moderate	Volume is very high (millions/day)
Per-call cost of $0.001–$0.01 is acceptable	Per-call cost must be minimal
Context-dependent judgment required	The task is well-defined and repeatable
A working prototype is needed quickly	Can invest weeks+ in model building

Table 8: Choosing between LLM-based and traditional ML classification. The two approaches are not mutually exclusive; a common pattern is to prototype with an LLM and migrate to a traditional classifier once labeled data accumulates and volume increases.

The LLM approach is analogous to hiring a talented generalist consultant: available immediately, capable of handling ambiguous situations with good judgment, but commanding a higher hourly rate. The traditional ML approach is more like training a dedicated specialist: it takes longer to ramp up, but once operational, the specialist is faster, cheaper per task, and highly reliable within a defined scope.

Step 3: Build and Test. Because the LLM-based approach requires no labeled training data and no model training infrastructure, it is considerably more accessible to non-specialists and is increasingly the default starting point for new classification projects. The remainder of this section focuses on the LLM workflow.

Drafting the prompt. A well-designed classification prompt has four components. The first is a *role assignment* that establishes context (e.g., "You are a customer support ticket classifier for an e-commerce company."). The second is a set of *category definitions* that describe each category with clear boundaries—not just the label, but what belongs and what does *not* belong. For instance:

> "`Billing`: Issues related to charges, refunds, payment methods, or invoice disputes. Does NOT include questions about pricing or subscription plans (those are `Account Management`)."

The third component is *few-shot examples*: two or three examples per category showing an input text and the correct label, chosen to illustrate boundary cases rather than obvious ones. The fourth is an *output format specification* that tells the model exactly what structure to return—for example, a JSON object containing a `category` field and a `confidence` field.[20]

Creating a test set. Before evaluating the classifier, a test set of 50–200 representative texts should be assembled. Each text in the set is labeled by a human reviewer who reads it and assigns the correct category—these are called *ground-truth labels* because they represent the known right answer against which the model's predictions will be judged. Ideally, each text is labeled independently by two or more reviewers; cases where reviewers disagree often reveal ambiguities in the taxonomy itself and are worth examining closely.

Systematic evaluation. Every text in the test set is run through the classifier, and predicted labels are compared against ground truth. The evaluation metrics described in the next section—precision, recall, and F1—are essential at this stage.

Iterating on the prompt. Error analysis drives improvement. If certain categories are systematically confused, the category definitions in the prompt need sharper boundaries. If edge cases are handled poorly, additional clarifying examples or explicit boundary rules can be added. This cycle of evaluate-analyze-revise repeats until performance meets the required threshold.

18.2 Evaluation Metrics: Measuring What Matters

Accuracy ("what percentage did the model get right overall?") is the metric most people reach for first, but it can be deeply misleading. Understanding **precision**, **recall**, and the **F1 score** is essential for any classification deployment.

[20] JSON (JavaScript Object Notation) is a lightweight, human-readable format for representing structured data as key-value pairs. A simple example: `{"category": "Billing", "confidence": "high"}`. JSON is the most common format for exchanging structured data between software systems and is natively understood by all major LLMs.

Precision measures the reliability of positive predictions. Of all the items the model labeled as category X, what fraction actually belong to category X?

High precision matters when false positives are expensive. For example, a fraud detection system that flags legitimate transactions as fraudulent creates terrible customer experience. You want high precision—when the system says "fraud", it should almost always be right.

Recall measures the completeness of detection. Of all the items that truly belong to category X, what fraction did the model correctly identify?

High recall matters when false negatives are expensive. For example, a compliance monitoring system that misses actual violations exposes the firm to regulatory risk. You want high recall—you need to catch (nearly) every real issue, even if that means flagging some false alarms.

F1 Score is the harmonic mean of precision and recall—a single number that balances both. It is useful as a summary metric, but the real insight comes from understanding the trade-off between precision and recall, not from optimizing a single number.

Think of precision and recall as a search party looking for a lost hiker. Recall is the probability that you find the hiker (you don't want to miss them). Precision is the fraction of areas you search that actually contain the hiker (you don't want to waste resources searching empty areas). A search party that checks every square mile of a 10,000-mile area has perfect recall but terrible precision. A search party that only checks the hiker's last known coordinates has great precision but may have low recall if the hiker has moved. The question is *what is more costly—a false alarm or a miss?* The answer determines whether you optimize for precision or recall.

18.3 Confidence Thresholds: The Automation Dial

Most classifiers can output not just a predicted label but also a *confidence score*—a measure of how certain the model is about its prediction. This score creates a powerful operational lever: by setting *confidence thresholds*, an organization can control the boundary between automated action and human review.

The basic pattern divides predictions into three bands. When confidence is high (e.g., above 0.9), the system auto-processes the text—routing the ticket, applying the label, or triggering the downstream action without human involvement. When confidence is moderate (e.g., 0.6–0.9), the model's prediction is surfaced as a suggestion for a human reviewer to confirm or override. When confidence is low (e.g., below 0.6), the text is routed to full human review with no model recommendation attached.

The confidence threshold is the single most important operational parameter in a classification deployment. Setting it is not a one-time event—it requires calibration against real data and ongoing adjustment.

Calibration proceeds in three steps. First, the classifier is run on a labeled test set and the confidence score for each prediction is recorded. Second, for each candidate threshold value (0.5, 0.6, 0.7, 0.8, 0.9), three quantities are calculated: the percentage of items that would be auto-processed, the accuracy among those auto-processed items, and the number of errors that would slip through without human review. Third, the threshold that best balances automation rate against acceptable error rate is selected, given the specific business context.

Example: A support ticket router operating at threshold 0.85 auto-routes 72% of tickets with 97% accuracy. Lowering the threshold to 0.7 increases auto-routing to 88% but drops accuracy to 91%. Whether the additional 16 percentage points of automation justify the accuracy reduction depends on the relative cost of misrouting a ticket versus the cost of human review time.

18.4 Multi-Label vs. Multi-Class Classification

A distinction that matters operationally is the difference between multi-class and multi-label classification. In *multi-class classification*, each text receives exactly one label from a set of mutually exclusive categories; for example, routing a support ticket to exactly one team.

In *multi-label classification*, each text can receive no, one, or multiple labels simultaneously; for example, tagging a product review with all applicable themes such as "shipping," "product quality," "packaging," and "customer service." Multi-label classification is common in practice but requires a different evaluation approach. Precision and recall must be measured *per label*, because a model might perform well on frequently discussed themes (e.g., "product quality") while performing poorly on rarer ones (e.g., "packaging"). Aggregating into a single accuracy number would mask these per-label disparities.

18.5 Monitoring in Production: The Feedback Loop

Deploying a classifier is not the finish line. Production monitoring is essential because customer language evolves, products are updated, new issue types emerge, and model performance can degrade over time.

Four signals are particularly informative. The first is *distribution shift*: if the proportion of tickets predicted as "Billing" suddenly jumps from 15% to 30%, the cause could be a genuine trend in customer issues or a model malfunction, and distinguishing between the two requires investigation. The second is the *confidence distribution*: a declining average confidence score suggests the model is encountering texts that fall outside the patterns it was designed for. The third is the *human override rate*: if reviewers override 25% of the model's medium-confidence predictions, the prompt or taxonomy likely needs revision. The fourth is *growth in the "Other" category*, which often signals that an emerging issue type deserves its own dedicated category.

Together, these signals form a feedback loop. Override decisions and distribution shifts surface weaknesses in the current system, which inform prompt revisions, taxonomy updates, and—if the traditional ML approach is in use—model retraining.

19 Information Extraction

If classification answers "What *type* of document is this?", information extraction answers "What specific *facts* are inside this document?" The goal is to convert free-form text into structured, database-ready records (Jurafsky and Martin, 2024, Chapter 17).

Consider, for example, a commercial real estate firm receiving hundreds of lease agreements per month, each in a different format from a different law firm. To populate their portfolio management system, an analyst must read each lease and key in the tenant name, property address, lease start and end dates, monthly rent, escalation clauses, renewal options, and any special conditions. This can take an experienced analyst over 30 minutes per lease. An information extraction system can do the first pass in seconds (Wei et al., 2023).

19.1 Designing the Extraction Schema

The **extraction schema** defines what fields to pull out of the text, what data types they should conform to, and how edge cases should be handled.

Schema Design Principles. The most important principle is to start with the downstream use case—the fields that a database, dashboard, or workflow requires—rather than by an attempt to extract everything that might be interesting. Extracting unnecessary fields adds cost, increases errors, and clutters the output.

Each field in the schema should be defined precisely. A field definition includes a name, a plain-language description of what it represents (e.g., "`lease_end_date`: The date on which the lease term expires, not including renewal options"), a data type (string, date, number, currency, boolean, enum, or array), whether the field is required or optional, and instructions for handling ambiguity (e.g., "If multiple dates appear, extract the one labeled 'Expiration Date' or 'Term End' "). This level of specificity may seem excessive, but vague field definitions are one of the most common sources of extraction errors.

Two further principles round out the design. *First*, missing information should be handled explicitly: the model should be instructed to return `null` or a designated sentinel value when a field is not present in the document, rather than guessing. A confidently fabricated value is worse than an explicit gap, because it enters downstream systems undetected. *Second*, confidence indicators should be included in the output: for each extracted field, the model reports a confidence level (high, medium, or low), enabling downstream systems to flag low-confidence extractions for human review.

The following JSON object illustrates a schema for extracting key fields from commercial lease agreements. Each entry defines a field name, its expected data type, whether it is required, and (where helpful) a description that guides the model on what to extract.

Example Schema: Commercial Lease Extraction

```
{
  "tenant_name": {"type": "string", "required": true},
  "landlord_name": {"type": "string", "required": true},
  "property_address": {"type": "string", "required": true},
  "lease_start": {"type": "date", "format": "YY-MM-DD", "required": true},
  "lease_end": {"type": "date", "format": "YY-MM-DD", "required": true},
  "rent": {"type": "currency", "currency": "USD", "required": true},
  "rent_escalation": {"type": "string", "required": false,
      "description": "Annual rent increase terms, e.g.,
      '3% annually' or 'CPI-linked'"},
  "renewal_opt": {"type": "array", "items": "string", "required": false},
  "special_cond": {"type": "array", "items": "string", "required": false}
}
```

19.2 The Extraction Pipeline

A production extraction system is more than a single LLM call. It is a multi-stage pipeline in which preprocessing, extraction, validation, and human review each play a role.

Stage 1: Document Preprocessing. Raw documents arrive in many formats—PDF, Word, scanned images, emails, HTML—and must be converted to clean text before extraction can begin. This seemingly mundane step is often where the most time is spent.

Text-based PDFs can be parsed directly, but scanned PDFs require Optical Character Recognition (OCR), and table-heavy PDFs require specialized table extraction tools. Documents with complex layouts (multi-column pages, headers and footers, sidebars) pose an additional challenge: the extraction tool must understand document structure to avoid jumbling content from different sections. Once raw text is obtained, a cleaning pass removes boilerplate text (i.e., letterheads, page numbers, repeated footers), corrects common OCR errors, and normalizes formatting inconsistencies.

The adage *garbage in, garbage out* is especially true for extraction. If OCR misreads "$1,200,000" as "$1,200.000" or truncates a key clause, no downstream model can recover the lost information. Investing in preprocessing quality before optimizing the extraction model almost always yields a higher return.

Stage 2: Extraction. The preprocessed text is passed to an LLM along with a prompt that includes the schema definition, extraction instructions, and—ideally—a few examples of correct extraction from similar documents. The model returns a structured object (typically JSON) conforming to the schema.

Stage 3: Validation. The raw extraction output must be validated before it enters any production system. Validation operates at several levels.

- *Format validation* checks whether each field conforms to its declared type: dates must

parse as valid dates, currency fields must be numeric, and the JSON structure itself must be well-formed.

- *Business rule validation* applies domain logic: a lease end date should fall after the start date, and a monthly rent figure should be within a plausible range for the property type and market.
- *Cross-field consistency* checks verify that fields agree with one another; for instance, if the lease term is stated as "5 years" and the start date is 2024-01-01, the end date should be 2028-12-31.
- A*completeness check* confirms that all required fields are populated and flags any suspiciously empty optional fields.

Items that fail any validation step are routed to human review.

Stage 4: Human Review. Even a 95%-accurate extraction system produces errors on 5% of documents. At a volume of 200 documents per month, that translates to roughly 10 documents with at least one incorrect field. The system should surface these exceptions for human review, prioritized by a combination of the model's confidence score and the impact of the affected fields; e.g., an error in `monthly_rent` is more consequential than a missing `special_conditions` entry.

19.3 Entity Resolution: The Hidden Data Quality Challenge

Once structured data has been extracted from many documents, a new problem emerges: the same real-world entity often appears under different names (Christophides et al., 2021). *Entity resolution* (also called record linkage or deduplication) is the process of determining which extracted names refer to the same real-world entity and merging them into a single canonical record. The problem is pervasive: a commercial lease corpus might yield the tenant names "Acme Corp", "Acme Corporation", "ACME Corp.", and "Acme

Corporation, Inc."—four surface-level variants that all refer to the same company. If a portfolio management system creates four separate tenant records, downstream analytics fragment: total leased square footage, revenue concentration, and renewal risk are all underreported for what is actually the largest tenant in the portfolio.

Three approaches are common in practice, each with different trade-offs. *Rule-based resolution* normalizes common variations (removing punctuation, standardizing abbreviations, expanding legal suffixes) and matches on the normalized strings. It is simple, fast, and effective for relatively clean data. *Embedding-based resolution* converts entity names to embedding vectors and flags pairs whose cosine similarity exceeds a threshold for review. This approach is better at catching non-obvious matches (e.g., "JPMorgan Chase" and "J.P. Morgan") because it operates on meaning rather than string similarity. *LLM-assisted resolution* presents candidate pairs to a language model and asks whether they refer to the same entity, taking into account abbreviations, legal suffixes, and common naming conventions. It is the most flexible approach but also the most expensive per comparison, making it best suited as a second-pass method applied to ambiguous cases that the cheaper approaches could not resolve.

19.4 Beyond Single Documents: Extraction at Scale

When extraction is applied across large document collections rather than one document at a time, several additional patterns become important.

The first is *template detection*. Documents originating from the same source (e.g., leases drafted by the same law firm) tend to follow a consistent structure. Detecting these templates makes it possible to optimize extraction prompts per template type, significantly improving accuracy on repeat formats. The second pattern is *incremental extraction*. When a contract is amended, only the changes need to be extracted and merged with the original record; re-extracting the entire document from scratch is wasteful and introduces unnecessary error risk. The third is *cross-document extraction*. Some facts are distributed across

multiple related documents: a master lease agreement defines base terms, while individual amendments modify rent, dates, or conditions. The extraction system must maintain linked records that reflect the cumulative state of the agreement, not just the content of any single document in isolation.

20 Semantic Search

Traditional search engines work by matching keywords. A search for "employee termination policy" returns documents containing those specific words. This approach, known as *keyword search*, is fast, well-understood, and powers the vast majority of search systems in use today.

Keyword search works remarkably well for many queries, particularly those involving proper nouns, product names, error codes, case numbers, or other precise identifiers. When someone searches for "SEC Form 10-K Acme Corp 2024", keyword matching finds exactly what is needed. This is one reason that Google and other major web search engines continue to rely heavily on keyword-based retrieval as a core component of their systems.

The limitations of keyword search become apparent when the query and the relevant documents use different words to express the same idea. This produces two types of failure. The first is *missed relevant documents* (false negatives): a document titled "Involuntary Separation Procedures" describes exactly the policy in question, but keyword search will not find it because it does not contain the word "termination". The second is *irrelevant results* (false positives): a document about "termination of the software license agreement" matches the keyword "termination" but has nothing to do with the HR query.

These failures are not edge cases—they are systematic. In any domain with diverse terminology, synonyms, jargon, or multilingual content, keyword search produces both gaps and noise. Semantic search addresses this by matching *meaning* rather than *words*, using the text embeddings introduced in Section 17 (Karpukhin et al., 2020).

20.1 The Semantic Search Pipeline

Semantic search uses embedding vectors to represent both documents and queries in a shared numerical space where proximity corresponds to similarity of meaning. The sys-

tem operates in two phases.

Phase 1: Indexing. The indexing phase is performed once and updated as documents are added, modified, or removed. First, each document in the corpus is split into smaller segments called *chunks* (chunking is discussed in detail below). Each chunk is then passed through an embedding model, producing a numerical vector that captures its meaning. Finally, the vectors are stored along with the original text and metadata such as the source document title, page number, and date in a *vector database* – a specialized system designed for fast similarity search over high-dimensional vectors.

Phase 2: Querying. When a search query arrives, it is embedded using the same model that was used during indexing, producing a query vector in the same numerical space. The vector database then identifies the stored vectors most similar to the query vector (using cosine similarity or a related measure) and returns the corresponding text chunks, ranked by similarity score.

The result is that a query for "employee termination policy" now retrieves the "Involuntary Separation Procedures" document, because both texts occupy the same region of the embedding space despite sharing no keywords.

20.2 Chunking: The Most Underrated Design Decision

Chunking—how documents are split into segments for embedding—has an outsized impact on search quality. It is routinely underestimated and difficult to get right.

An embedding vector represents the *overall meaning* of its input text. If a chunk is too large (e.g., an entire 30-page document), the embedding becomes a blurry average of many topics and will not match specific queries well. If a chunk is too small (e.g., a single sentence), it may lack the context needed to be meaningful or useful as a search result. The goal is segments that are focused enough to match precise queries yet long enough

to carry sufficient context.

Chunking Strategies. Several approaches are in common use, each suited to different document types.

Strategy	How It Works	Best For
Fixed-size	Split every N tokens with overlap between adjacent chunks	Simple baseline; homogeneous documents
Paragraph	Split on paragraph boundaries	Well-structured documents with natural paragraph breaks
Section	Split on headings / section markers	Reports, manuals, and policies with clear sections
Semantic	Detect topic shifts algorithmically and split at natural boundaries	Documents with varied structure; highest quality but most complex
Recursive	Start with large chunks; split if a size limit is exceeded, respecting sentence and paragraph boundaries	General-purpose default; good balance of simplicity and quality

Table 9: Common chunking strategies for semantic search indexing. In practice, recursive chunking is the most widely used default.

Regardless of the strategy chosen, overlap between adjacent chunks is important. If a relevant passage straddles a chunk boundary, overlap ensures it appears in full in at least one chunk. A typical overlap is 10–20% of the chunk size.

For most applications, a reasonable starting point is chunks of 256–512 tokens with 50–100 tokens of overlap. The next step is to test with representative queries and inspect the results manually: if results feel too narrow or lacking in context, chunk size should be increased; if results feel diluted or off-topic, it should be decreased.

A useful technique is to decouple retrieval granularity from display granularity. The idea is straightforward: embed and retrieve *small* chunks (which match queries precisely), but when displaying results to the user, show the *larger surrounding context* so that the result is actually useful to read.

For example, consider a company's employee handbook that is 80 pages long. For retrieval purposes, the system chunks it into focused passages of, say, 300 tokens each.

When an employee searches for "how many vacation days do new hires get," the retrieval step identifies a small chunk that states "Employees in their first year of service accrue 15 days of paid time off." That chunk matched the query well precisely because it was small and topically focused. But if the search interface displayed only that one sentence, the employee would lack important context: What about employees after their first year? Is there a cap on accrual? Are there blackout dates? Rather than showing the isolated chunk, the system displays the full "Paid Time Off" section of the handbook—the *parent* of the retrieved chunk—giving the employee enough surrounding context to get a complete answer. The small chunk determined *what* to retrieve; the larger section determined *what to show*.

This pattern avoids a difficult trade-off. Without it, the system designer must choose a single chunk size that serves both retrieval accuracy and display usefulness—competing goals that pull in opposite directions. Decoupling the two makes it possible to optimize each independently.

20.3 Vector Databases: The Infrastructure Layer

A vector database is purpose-built for storing and searching embedding vectors at scale. It can be thought of as a library where books are shelved not alphabetically or by call number but by conceptual similarity—books about related topics sit on the same shelf, regardless of their titles or authors.

A naive approach to vector search—comparing the query vector to every stored vector one by one—would be prohibitively slow for large collections. For example, a corpus of one million document chunks, each represented by a 1,536-dimensional vector, would require computing one million cosine similarity scores per query. At ten million or one billion chunks, brute-force comparison becomes impractical. Vector databases solve this problem using algorithms called *approximate nearest neighbor (ANN)* methods, which build an index structure that enables near-instant retrieval with only a minor trade-off in recall

(Johnson, Douze and Jégou, 2021).

The intuition behind ANN algorithms is analogous to how a well-organized library works. Brute-force search means walking through every aisle and inspecting every book. An ANN index groups related vectors into neighborhoods during the indexing phase, so that at query time the system navigates directly to the most promising neighborhoods and searches only there. A small number of relevant results may be missed (such as a book being mis-shelved in the wrong section), but 95–99% of the best matches are found in a fraction of the time.

Key Capabilities. Beyond fast similarity search, production vector databases offer several capabilities that matter for enterprise deployment. *Metadata filtering* allows semantic similarity to be combined with traditional structured filters; e.g., retrieving documents similar to a query *and* published after 2023 *and* tagged as "legal". The vector database applies both constraints simultaneously, avoiding the need to filter results after retrieval. Scalability is another critical property: production-grade systems handle billions of vectors with sub-second query latency. Finally, -time update support means that new documents can be embedded and added to the index without rebuilding it from scratch, which is essential for corpora that change frequently.

20.4 Hybrid Search: Combining Keywords and Meaning

In practice, neither pure keyword search nor pure semantic search is optimal for all queries. Each has systematic strengths and weaknesses, and a well-designed search system combines both.

Keyword search excels on *precise, unambiguous queries* where the user knows the right terms: a product SKU, a regulation number, an employee ID, a specific error message. It is also very fast and computationally cheap. But it fails on queries that require understanding intent or that use different vocabulary than the target documents.

Semantic search excels on *conceptual, natural-language queries* where the user describes a need rather than naming a specific item. But it has its own weaknesses. Embedding models can sometimes place semantically different texts too close together (e.g., "increase revenue" and "decrease revenue" may have similar embeddings because they share most of their semantic content, differing only in direction). Semantic search can also struggle with rare proper nouns or technical identifiers that the embedding model has not encountered frequently enough to represent well.

Hybrid search combines both approaches to capture the strengths of each. The system runs a keyword search and a semantic search in parallel, then merges and re-ranks the two result sets (more on "re-ranking" in the next subsection).

As an example, consider an internal search system at a financial services firm. An analyst searches for "FINRA Rule 3110 supervisory obligations". The keyword component retrieves documents that literally mention "FINRA Rule 3110"—the exact regulatory text, internal compliance memos referencing the rule by number, and training materials with the rule in their title. The semantic component retrieves documents about supervisory review procedures, branch office oversight policies, and trade surveillance workflows—documents that discuss the *substance* of the rule without necessarily citing it by number. The merged result set includes both: the precise regulatory references *and* the operationally relevant internal documents, ranked in a way that surfaces the most useful results first. Neither keyword nor semantic search alone would have produced this complete picture (Cormack, Clarke and Buettcher, 2009).

Hybrid search is most valuable in corpora that contain a mix of structured identifiers and natural-language discussion—which describes most enterprise knowledge bases. It is less critical for corpora that are purely natural-language prose (where semantic search alone performs well) or purely structured records (where keyword or metadata filtering suffices).

Most modern vector databases (including Elasticsearch and Pinecone) support hybrid

search natively, making it a configuration choice rather than a custom engineering project.

20.5 Reranking: A Second Pass for Precision

Initial retrieval—whether keyword, semantic, or hybrid—is designed to be fast. It scans a potentially enormous corpus and returns a manageable candidate set (typically 20–100 results) in milliseconds. But speed comes at the cost of precision: the initial ranking is approximate, and the ordering of the top results is often imperfect.

Reranking adds a second, more computationally expensive pass that refines the ordering of the candidate set (Nogueira and Cho, 2019). A *reranker model* takes each candidate result paired with the original query and produces a relevance score that considers the fine-grained interaction between the query and the document. This is fundamentally different from the initial retrieval step, where the query and documents are embedded independently and compared only by vector distance.

The difference is analogous to two ways of evaluating job candidates. In the initial retrieval step (the "resume screen"), each candidate's resume is scored independently against a job description—fast, but unable to catch subtle mismatches. In the reranking step (the "interview"), each candidate is evaluated in direct conversation with the hiring criteria—slow, but far more accurate at distinguishing between a candidate who is superficially relevant and one who is genuinely the best fit.

Technically, the improvement comes from the reranker's architecture. Initial retrieval encodes the query and each document separately into fixed vectors, then compares them with a simple distance metric. The reranker processes the query and the document together as a single input, allowing the model to attend to fine-grained interactions—such as whether a specific clause in the document actually answers the specific question posed, or merely discusses a related topic.

For example, suppose an HR manager searches an internal knowledge base for "parental leave policy for employees in California". The initial retrieval returns 30 candidate chunks.

Several of these discuss parental leave policies for other states, and several discuss California-specific employment law but not parental leave. Both are semantically close to the query but do not answer it. A reranker, processing each chunk alongside the full query, can distinguish between a chunk that mentions California employment law in general and a chunk that specifically addresses California parental leave—a distinction that vector distance alone struggles to capture.

Because the reranker processes each candidate individually rather than performing a single index lookup, its cost scales linearly with the number of candidates. Processing 50 candidates through a reranker is practical; processing 10,000 is not. This is why the two-stage architecture matters: initial retrieval narrows the universe cheaply, and reranking applies expensive precision only to the short list.

In practice, reranking is one of the highest-impact improvements available when search quality is not meeting expectations. It is often more effective than switching to a better embedding model or redesigning the chunking strategy, because it addresses the ordering of the final results that users actually see.

20.6 Evaluating Search Quality

Subjective impressions ("the results look good") are insufficient for evaluating a search system. Rigorous evaluation requires quantitative metrics computed against a test set of queries with known relevant documents. Four metrics are widely used.

- *Precision@k* measures the fraction of the top k results that are actually relevant—in plain language, it answers the question "are the top results good?"
- *Recall@k* measures the fraction of all relevant documents that appear in the top k results, answering "are we missing anything important?"
- *Mean Reciprocal Rank (MRR)* captures how far down the result list a user must go, on average, to find the first relevant result—a measure of how quickly the system

delivers a useful answer.

- *Normalized Discounted Cumulative Gain (NDCG)* considers not only whether relevant documents appear in the results but whether the most relevant ones are ranked highest, penalizing systems that bury the best results below weaker ones (Järvelin and Kekäläinen, 2002).

For most business applications, Precision@5 or Precision@10 (depending on how many results the interface displays) and MRR are the most immediately informative metrics. NDCG is valuable when the use case involves browsing a ranked list rather than finding a single best answer.

A sound evaluation set is built in four steps. First, 30–50 representative queries are collected from actual or anticipated users. Second, for each query, a domain expert identifies the 3–10 most relevant documents (or passages) in the corpus. Third, each query is run through the search system and the results are scored against the expert judgments using the metrics above. Fourth, the evaluation is repeated after any change to the chunking strategy, embedding model, reranker, or retrieval parameters.

20.7 Applications

Semantic search delivers the greatest value in domains where terminology is varied, jargon is common, or users cannot be expected to know the exact words used in the target documents. For example:

- Internal knowledge bases are the most common enterprise use case, enabling employees to find relevant policies and procedures even when they do not know the organization's exact terminology.

- Customer self-service portals benefit similarly, matching support articles to customer intent rather than requiring customers to use the right keywords, which reduces support ticket volume.

- Legal discovery is a high-value application where attorneys can search for relevant precedents by legal concept rather than specific terms.
- Product catalog search allows shoppers to describe needs in natural language ("something to keep my coffee hot during my commute") rather than navigating a category hierarchy.
- Talent matching uses semantic similarity to connect job descriptions with candidate profiles based on skills and experience, moving beyond keyword-based resume scanning that misses qualified candidates who describe their experience differently.
- Research literature search helps scientists find related work across different naming conventions, subfields, and languages—a domain where vocabulary fragmentation is severe.

21 Retrieval-Augmented Generation (RAG)

LLMs are trained on public data up to a certain cutoff date. They have no access to an organization's internal documents, latest financial results, proprietary research, or customer records. When asked about information outside their training data, they either acknowledge ignorance or—more dangerously—produce a confident-sounding but fabricated answer. This phenomenon is called a *hallucination*, and it is one of the most important risks in deploying LLMs (Huang et al., 2023).

Retrieval-Augmented Generation (RAG) addresses this by giving the LLM access to private data *at query time*, without retraining the model (Lewis et al., 2020). It is the most widely deployed means for building enterprise question-answering systems with LLMs.

RAG is analogous to a brilliant consultant who has just been hired and knows nothing about the company. Asking the consultant to answer based on general knowledge would produce confident guesses—some right, many wrong. The better approach is to hand the consultant the relevant files and say, "Read these first, then answer my question based on what is in these documents."

For a comprehensive treatment of RAG architectures, retrieval strategies, and evaluation, see Gao et al. (2024).

21.1 The RAG Pipeline

A production RAG system operates in two major phases: an *ingestion phase* that builds a searchable knowledge base (done once and updated as documents change), and a *query-and-response phase* that handles each user question.

Phase 1: Ingestion. The ingestion phase converts a collection of source documents into an embedding index that can be searched semantically. It proceeds through five steps, each involving design decisions with meaningful downstream consequences.

The first step is *source identification*: determining which documents should feed the knowledge base. This is a design choice with significant implications. If the scope is too narrow, the system will be unable to answer many legitimate questions. If too broad, irrelevant context may confuse the model or dilute retrieval quality. Common sources include internal wikis, policy documents, product documentation, FAQ databases, support ticket archives, and regulatory filings.

The second step is *document preprocessing*: extracting clean text from source formats such as PDF, Word, and HTML. Tables, images with captions, code blocks, and other non-prose content require special handling. This is the same preprocessing challenge discussed in the information extraction module (Section 19), and the same principle applies: preprocessing quality sets a ceiling on everything downstream.

Third, the cleaned text is *chunked* into segments suitable for embedding. The chunking strategies described in Section 20 apply here, with one additional constraint: retrieved chunks must fit within the LLM's context window alongside the user's question and the generation instructions. If chunks are 500 tokens each and 10 are retrieved, that consumes 5,000 tokens of context before the system prompt and user question are even included.

Fourth, useful *metadata* is attached to each chunk: the source document title, page number, section heading, document date, document type, and access permissions. This metadata serves two purposes. At query time, it enables filtering (e.g., restricting retrieval to documents from the current year or a specific department). In the generated answer, it enables *source attribution*—the ability to cite where each piece of information came from, which is essential for user trust and auditability.

Finally, each chunk is converted to an embedding vector and stored in a vector database, completing the index.

Phase 2: Query and Response. When a user submits a question, the system executes a sequence of steps that connect retrieval to generation.

The question is first *processed* and optionally transformed to improve retrieval quality (advanced transformation strategies are discussed below). The processed query is then *embedded* using the same model that was used during ingestion, and the vector database returns the k most similar chunks—typically $k = 5$–20, depending on chunk size and the LLM's context window capacity.

If a reranker is available, the retrieved chunks are *reranked*: this entails scoring each chunk for relevance to the specific query and reordering the chunks accordingly. As discussed in Section 20, this step often produces a meaningful improvement in the quality of the final answer because it evaluates the fine-grained interaction between the query and each candidate chunk.

The reranked chunks are then assembled into a *prompt* that instructs the LLM. A typical system prompt reads:

> *"Based on the following context documents, answer the user's question. Cite the source document for each claim. If the answer is not fully supported by the provided context, explicitly state what is uncertain. Do not use information from outside the provided context."*

The wording of this system prompt is one of the most consequential design decisions in a RAG system. It governs hallucination risk (how likely the model is to invent information), citation behavior (whether and how sources are referenced), and ambiguity handling (what the model does when the retrieved context is incomplete or contradictory). Like classification prompts, the system prompt should be treated as an engineering artifact: versioned, tested against the evaluation set, and revised based on observed failure patterns.

The LLM then *generates* a natural-language answer grounded in the retrieved context, ideally with inline citations. A final *post-processing* step parses citations from the response, verifies that each citation references an actual retrieved chunk (rather than a fabricated source), and formats the answer for the user interface.

21.2 Advanced Retrieval Strategies

The basic approach—embed the query, retrieve the most similar chunks—works well for straightforward, clearly phrased questions. Real-world queries, however, are often vague, complex, or poorly worded. Several advanced strategies address these challenges.

Query Transformation. User questions frequently arrive in forms that do not retrieve well. A question like "What's our PTO policy?" is colloquial and underspecified; it may fail to retrieve chunks that discuss "paid time off accrual rates" or "vacation day carryover limits" because the embedding of the short, casual question is not close enough to the embeddings of the detailed policy language.

Query expansion addresses this by using the LLM to rewrite the query before retrieval. The original question "What's our PTO policy?" might be expanded to "What is the company's paid time off policy, including accrual rates, maximum carryover, and approval procedures?" The expanded version is more likely to land near the relevant chunks in embedding space.

A more sophisticated variant is *Hypothetical Document Embeddings* (HyDE). Instead of expanding the query, the LLM generates a hypothetical *answer*—without access to any documents—and that hypothetical answer is embedded and used for retrieval. The logic is that a plausible answer, even if fabricated, is linguistically closer to the real document containing the real answer than the original question is. The hypothetical answer uses the same vocabulary, sentence structure, and level of detail that the target document is likely to use. HyDE often outperforms direct query embedding, particularly for questions that are phrased very differently from the language in the knowledge base (Gao et al., 2023).

Multi-Query Retrieval. A single query embedding may not capture all facets of a complex question. *Multi-query retrieval* addresses this by using the LLM to decompose the original question into several reformulated queries, running retrieval for each, and merg-

ing the result sets.

For example, the question "How does our parental leave policy compare to industry standard?" contains two distinct information needs: the company's own policy and external benchmarks. A single embedding of this question may land in between the two topics, retrieving neither well. Decomposing it into "company parental leave policy details," "tech industry parental leave benchmarks," and "maternity and paternity leave comparison" allows each sub-query to retrieve its own relevant chunks, and the merged result set covers both facets of the original question.

Parent-Child Chunk Architecture. This strategy addresses the tension between retrieval precision and generation context. Small chunks retrieve precisely but may lack the surrounding context the LLM needs to generate a complete answer. Large chunks provide rich context but match queries less precisely.

The parent-child architecture resolves this by maintaining two levels of chunking. Small *child* chunks (e.g., 200 tokens) are embedded and indexed for retrieval. Each child chunk is linked to a larger *parent* chunk (e.g., the full section or page it came from). When a child chunk is retrieved, its parent is sent to the LLM as context. This gives the retrieval step the precision of small chunks while giving the generation step the contextual richness of large ones. The concept is identical to the retrieval-display decoupling described in Section 20, applied here to the generation step rather than to a search interface.

21.3 Evaluating RAG Systems

RAG evaluation is more complex than evaluating a search system or a standalone LLM, because there are two independent dimensions that can each succeed or fail: *retrieval quality* (did the system find the right documents?) and *generation quality* (did the LLM produce a good answer from those documents?). A system can retrieve the right documents but generate a poor answer, or generate a fluent answer from the wrong documents. Both

failure modes must be measured separately.

Dimension	Metric	What It Measures
Retrieval	Recall@k	Of the documents containing the answer, how many appear in the top k?
Retrieval	Precision@k	Of the k retrieved documents, how many are relevant?
Generation	Faithfulness	Does the answer reflect the retrieved context without adding fabricated claims?
Generation	Relevance	Does the answer actually address the question that was asked?
Generation	Completeness	Does the answer cover all aspects supported by the retrieved context?
End-to-end	Correctness	Is the final answer factually correct? (The ultimate metric; hardest to measure.)

Table 10: Evaluation metrics for RAG systems, organized by the dimension they assess (Es et al., 2024).

The most diagnostic pattern in RAG evaluation is to measure retrieval and generation metrics *independently*. If retrieval recall is high but answer correctness is low, the problem lies in the generation step (the prompt, the model, or how context is presented). If retrieval recall is low, no amount of prompt engineering will fix the answers—the retrieval pipeline needs attention first. This separation prevents the common mistake of blaming the LLM for errors that originate in retrieval.

Building a RAG Evaluation Set

A RAG evaluation set is built by collecting 50–100 representative questions that real users would ask, then identifying for each question (a) the specific documents or passages that contain the answer and (b) a human-written reference answer. Each question is run through the full pipeline, retrieval metrics are computed against the known source documents, and generated answers are evaluated against the reference answers—either by human reviewers or by using an LLM as an evaluator with detailed scoring rubrics. Metrics should be tracked over time as chunking, retrieval, and prompts are modified.

This evaluation set is the most important artifact in a RAG project. It is what makes it possible to compare design alternatives, diagnose failures, and make data-driven decisions rather than relying on anecdotal impressions of answer quality.

21.4 Common Failure Modes & Mitigations

Six failure modes account for the majority of RAG system issues in production.

The most common is *wrong documents retrieved*: the retriever finds chunks that are topically plausible but do not contain the correct answer, and the LLM generates a confident response based on this misleading context. Mitigations include improving the chunking strategy, adding hybrid search, introducing reranking, and expanding the evaluation set to cover more query types.

The second failure mode is the *answer not in the knowledge base*: the user asks a question that the documents simply do not cover. Without explicit instruction, the LLM may hallucinate an answer rather than admitting the gap. The system prompt should instruct the model to state clearly when the retrieved context does not support an answer, and confidence thresholds can be used to flag low-confidence responses for human review.

Third, *contradictory sources* arise when different documents say different things—for example, an outdated policy memo that conflicts with a recently updated handbook. If the model encounters conflicting information, it may arbitrarily choose one version. The system prompt should instruct the model to surface contradictions explicitly, and metadata (particularly document dates) should be included in the context so the model can prefer more recent sources.

Fourth, *context window overflow* occurs when the volume of retrieved chunks exceeds the LLM's context limit. Mitigations include limiting k, using reranking to prioritize the most relevant chunks, and—in extreme cases—summarizing retrieved content before passing it to the model.

Fifth, *stale information* results from outdated documents remaining in the knowledge

base. This is an operational rather than a modeling problem: document refresh schedules, version tracking, and displaying document dates in generated answers all help users assess currency.

Sixth, *citation errors* occur when the model attributes a claim to the wrong source document or fabricates a citation entirely. Post-processing logic should verify that every citation in the generated answer maps to an actual retrieved chunk; citations that cannot be verified should be flagged or removed.

21.5 RAG vs. Fine-Tuning

A question that frequently arises in executive discussions is why the LLM cannot simply be "trained on our data" rather than building a retrieval pipeline. The answer lies in what each approach actually does. RAG gives the model access to *knowledge*—facts, documents, data—at query time. Fine-tuning changes the model's *behavior*: its style, output format, reasoning patterns, or task-specific capabilities. They solve different problems.

In practice, most enterprise LLM deployments use RAG, because the dominant need is to ground the model in private, frequently changing data—not to change how the model writes or reasons. That said, the two approaches are not mutually exclusive—an organization might fine-tune a model for a particular output style and simultaneously use RAG to ground it in current data—but RAG is nearly always the starting point.

22 Text Clustering

Unsupervised learning techniques such as k-means and hierarchical clustering were introduced in Part II using tabular data: each observation is described by numerical features like age, income, or purchase frequency, and the algorithm groups similar observations together without predefined labels. Text clustering applies the same algorithms to a different input representation (Grootendorst, 2022). Instead of tabular feature vectors, each document is represented by its *embedding vector*—the numerical representation of meaning introduced in Section 17.

The fundamental question is identical: can natural groupings be discovered in a dataset without predefined labels? The difference is that the "features" are now embedding dimensions that capture semantic content rather than demographic or transactional attributes. A customer review about slow shipping and a customer review about late delivery will land near each other in embedding space—and therefore in the same cluster—even if they share no words in common.

22.1 The Text Clustering Workflow

Step 1: Embed. The first step is to convert each text (a document, paragraph, or sentence, depending on the application) into an embedding vector. Two decisions matter here.

The first is *granularity*: customer reviews typically cluster well at the document level because each review discusses a single experience, while long research reports may require paragraph-level embedding to capture multiple topics within a single document.

The second is *model consistency*: all texts in a clustering project must be embedded with the same model, because vectors from different models occupy different numerical spaces and cannot be meaningfully compared.

Step 2: Reduce Dimensions. Embedding vectors have hundreds or thousands of dimensions. Clustering algorithms struggle in very high-dimensional spaces due to a phenomenon known as the *curse of dimensionality*: as the number of dimensions grows, the distances between all pairs of points converge toward the same value, making it progressively harder to distinguish clusters from noise. Dimensionality reduction techniques project the high-dimensional vectors into a lower-dimensional space—typically 2 to 50 dimensions—while preserving the relationships between points that matter for clustering. This step is usually essential, not optional.

A common and effective approach is a two-stage pipeline: Principal Component Analysis (PCA) first (reducing from 768 dimensions to roughly 50), then UMAP (reducing from 50 to the final target). This sequence is faster and often produces better-separated clusters than applying UMAP directly to the full-dimensional vectors, because PCA removes noise that can interfere with UMAP's neighborhood calculations.

Step 3: Cluster. A clustering algorithm is applied to the reduced-dimensional embedding vectors. Four algorithms are in widespread use, each with distinct trade-offs.

Algorithm	**Strengths**	**Weaknesses**
K-Means	Simple, fast, scales well, widely understood	Must specify k upfront; assumes similar-sized spherical clusters
HDBSCAN	Finds clusters of varying density; does not require specifying k; identifies outliers	More parameters to tune; can be slow on very large datasets
Hierarchical	Produces a dendrogram showing relationships at multiple granularities	Slow for large datasets; requires choosing where to cut the tree
GMM	Each point receives a probability of belonging to each cluster	More complex to configure; assumes Gaussian distributions

Table 11: Clustering algorithms commonly used for text data.

K-Means is a reasonable starting point when there is a rough expectation of how many clusters to look for. It is fast, intuitive, and produces clean cluster boundaries. The elbow

method or silhouette score (discussed below) can guide the choice of k. HDBSCAN is preferable when the number of clusters is unknown or when clusters are expected to vary significantly in size and density. Its ability to label some points as "noise", not belonging to any cluster, is particularly valuable for messy real-world text data, where not every document fits neatly into a theme. When time permits, running both algorithms and comparing the results is informative: if K-Means and HDBSCAN produce broadly similar groupings, there is greater confidence that the structure is real rather than an artifact of the algorithm.

Step 4: Interpret and Label Clusters. Raw clusters are just numbered groups of vectors carrying no inherent meaning. The step that converts them into actionable insight is *labeling*: assigning a descriptive name to each cluster that captures its common theme. Several approaches are available.

Manual inspection involves reading 10–20 randomly sampled documents from each cluster and writing a label based on the common thread. It is the most accurate method but time-consuming, and it scales poorly beyond a dozen or so clusters.

LLM-assisted labeling passes a representative sample from each cluster to a language model with the instruction to identify the common theme and generate a concise label (Zhang et al., 2023). This approach is fast and surprisingly effective, particularly for an initial pass.

Keyword extraction uses statistical methods to identify the most distinctive words in each cluster and derives a label from the top terms; it is quick but sometimes produces labels that are more cryptic than informative.

In practice, a *hybrid* approach works best: LLM-generated labels provide a starting point, and domain experts refine them based on their knowledge of the business context. A genuine risk when interpreting clusters is confirmation bias: it is tempting to see patterns that align with the LLM-generated labels. A useful safeguard is to have someone

other than the analyst who performed the first step read a random sample from each cluster and independently propose a label. If the independent labels diverge significantly, the cluster may be less coherent than it first appeared, or the original label may be projecting structure that is not actually there.

22.2 Determining the Right Number of Clusters

Selecting k is one of the most common questions in any clustering project, and there is no single correct answer. Several tools help navigate the decision.

The *elbow method* plots total within-cluster variance against different values of k. As k increases, variance decreases (more clusters means tighter groupings), but at some point the marginal improvement diminishes sharply—the "elbow" of the curve. This inflection point suggests a natural number of clusters. The *silhouette score* measures how similar each point is to its own cluster compared to the nearest neighboring cluster, producing a value between -1 and $+1$ (higher is better). It is useful for comparing different values of k and for identifying clusters that are poorly separated.

Beyond these statistical tools, *business pragmatics* play an essential role. A statistically optimal $k = 47$ is not useful if the organization can only realistically act on 8–10 themes. A *hierarchical approach* is often practical: fist cluster at a granularity the team can absorb, then selectively split clusters that appear to contain multiple distinct sub-themes.

22.3 Temporal Clustering: Tracking How Themes Evolve

One of the highest-value applications of text clustering is tracking how themes change over time. By clustering the same data source at regular intervals—monthly customer feedback, weekly support tickets, quarterly survey responses—it becomes possible to detect patterns that a single point-in-time analysis would miss.

A new cluster appearing in March that was absent in February signals an emerging

issue or trend. A cluster that grows from 5% to 18% of support tickets over three months provides a quantifiable signal that something has changed. Conversely, if a product fix is shipped in April and the related complaint cluster shrinks in May, the clustering analysis provides evidence that the intervention worked.

The key technical challenge is making clusters comparable across time periods. Two approaches are common. In the first, each month (or quarter) is clustered separately, and the resulting clusters are matched across periods by asking which ones look most similar—much like comparing two independently drawn maps of the same territory. This approach is good at spotting genuinely new themes but can struggle to match clusters that shift gradually. In the second approach, all periods are pooled into a single dataset and clustered once, producing a fixed set of themes. The analysis then tracks how the share of documents in each theme changes over time. This is simpler but less sensitive to emerging themes, because a small new pattern may be absorbed into a larger existing cluster rather than surfacing on its own.

22.4 Applications

Text clustering sees widespread use across several functions.

Voice of the customer. Product reviews, support tickets, and open-ended survey responses are natural candidates for clustering. Star ratings and predefined categories capture expected themes but miss what customers talk about in their own words. Clustering frequently surfaces themes—packaging quality, onboarding confusion, accessory compatibility—that were never part of the original feedback taxonomy and would not have been discovered without an unsupervised approach.

As an example, consider a consumer electronics company that collects thousands of product reviews per quarter. Star ratings provide a coarse signal, and predefined survey categories capture expected themes, but neither reveals what customers are *actually* talking about in their own words. Embedding and clustering the review text often reveals

previously hidden information; e.g., that "packaging quality" is a distinct theme raised in many reviews—something never tracked in the company's feedback taxonomy. Such an insight may lead to a packaging redesign that improves satisfaction scores.

Employee engagement. Free-text responses in engagement surveys often contain richer signal than the structured questions they accompany. Clustering these responses regularly reveals themes that cut across predefined categories—cross-team collaboration barriers, tooling frustrations, or onboarding gaps—surfacing organizational issues that structured survey instruments were not designed to detect.

Take, for instance, an HR team analyzing free-text responses from an engagement survey. The predefined response categories ("compensation," "management," "work-life balance") capture the obvious themes, but clustering might reveal other latent themes—e.g., one around "cross-team collaboration barriers"—that structured surveys had missed entirely.

Competitive intelligence. News articles, press releases, and analyst commentary mentioning competitors can be clustered over time to track shifts in media coverage. Temporal clustering is especially valuable here: a new cluster emerging around "supply chain restructuring" or "regulatory scrutiny" can provide early warning of strategic moves before they are formally announced.

Consider, for example, a product marketing team that monitors blog posts, analyst reports, and social media commentary about its product category. Each month, the team embeds and clusters the latest batch of content. For most of the year, the clusters are familiar: pricing comparisons, feature wishlists, integration challenges, and migration case studies. Then in September, a new cluster appears around "AI-assisted workflow automation"—a theme absent from prior months. By November, the cluster has grown from 3% to 14% of all content. The marketing team now has a quantitative signal that the market conversation is shifting, and can reposition its messaging and product roadmap discussions accordingly—months before a formal analyst report crystallizes the trend into

a headline.

23 Compliance & Risk Detection

Every regulated industry faces a version of the same problem: enormous volumes of text that must be monitored for policy violations, and severe consequences for missing something. Financial services firms must surveil millions of communications for insider trading, market manipulation, and conduct violations. Healthcare organizations must review clinical documentation for billing compliance and patient safety signals. Insurance companies must analyze claims narratives for fraud patterns. Corporations of all kinds must monitor internal communications for harassment, discrimination, and code-of-conduct violations.

Before NLP, compliance monitoring relied on two imperfect approaches. The first was *keyword lexicons*: lists of suspicious words and phrases that triggered alerts whenever they appeared. The fundamental problem with lexicons is that language is ambiguous. The word "kill" in "we need to kill this project" is not a threat. The phrase "this is going to explode" in reference to a product launch is not a safety concern. Keyword-based systems generated overwhelming false positives, burying genuine issues in noise. The second approach was *random human sampling*: compliance officers manually reviewed a small, randomly selected fraction of communications. This was thorough for the items reviewed but statistically guaranteed that the vast majority of violations—which are rare events in a large communication stream—would go undetected.

Modern NLP-powered compliance systems address both failures by combining the speed and breadth of automation with the contextual judgment of human reviewers. See Bao et al. (2023) for a comprehensive survey.

23.1 A Layered Architecture

The standard design pattern for compliance monitoring is a *layered funnel* in which each successive layer narrows the volume of items under consideration while increasing the

depth of analysis.

Layer 1: Broad Screening. All communications pass through an initial NLP filter designed to catch anything potentially problematic. This layer is calibrated to maximize *recall*—the goal is to flag every item that might be an issue, accepting that many flagged items will turn out to be benign.

The screening layer typically combines several techniques in parallel. Keyword and phrase lexicons remain useful as a baseline, but they are now supplemented by text classification models trained to detect specific risk categories (insider trading language, harassment indicators, unauthorized disclosure patterns), information extraction to identify entities such as people, companies, securities, dates, and monetary amounts, and anomaly detection to flag communications that are statistically unusual—abnormal volume from a particular sender, messages sent at unusual times, or unexpected communication between individuals who do not normally interact.

The design principle at this layer is high recall: false positives are tolerable; false negatives are not.

Layer 2: Risk Scoring and Prioritization. Flagged items from Layer 1 are scored and ranked by risk level. This is where LLMs add the most value, because they can assess *context* that simpler models miss. An LLM can distinguish between "let's discuss this offline" said about a public company event (benign) and the same phrase said about a pending merger (warranting scrutiny). It can differentiate "I have material information" (potentially problematic) from "I'll send you the marketing materials and information" (benign). It can recognize coded or euphemistic language that evades keyword filters entirely.

Operationally, each flagged item receives a numerical risk score based on the LLM's assessment. Items above a high threshold (e.g., 80 out of 100) are escalated immediately. Items in a medium range (e.g., 50–80) enter a prioritized review queue. Items below the

lower threshold are archived with the flag notation but are not actively reviewed unless a broader investigation later pulls them in.

Layer 3: Human Review. The highest-risk items are routed to human compliance officers who evaluate the evidence and make a determination. The AI system triages and prioritizes; *humans make the final call*. This is the human-in-the-loop step, and in regulated environments it is non-negotiable.

This layered architecture is analogous to an emergency room triage system. Every patient who arrives receives an initial assessment at the front desk (Layer 1). Those with concerning vital signs receive further evaluation by a nurse (Layer 2). The most serious cases go directly to the attending physician (Layer 3). The system ensures no one is turned away, but the physician's limited time is focused where it matters most.

23.2 Threshold Tuning: The Central Operational Challenge

Setting the thresholds between layers is the single most consequential operational decision in a compliance system, because it directly governs the trade-off between two competing failure modes.

If thresholds are set too low (too sensitive), the system generates excessive false positives. Reviewers experience alert fatigue: overwhelmed by benign items, they begin rubber-stamping reviews rather than evaluating them carefully, which defeats the purpose of the human review layer entirely. If thresholds are set too high (too permissive), the system misses actual violations—false negatives—exposing the organization to regulatory fines, legal liability, and reputational damage.

Calibrating Compliance Thresholds. Threshold calibration typically proceeds in four stages. The first is to run the system on a historical dataset with known outcomes—past investigations that resulted in confirmed violations and confirmed non-issues—to

establish baseline false positive and false negative rates at various threshold values.

Second, the asymmetric costs of each error type have to be estimated. For example, each false positive might cost roughly one hour of reviewer time, while a single false negative—a missed insider trading violation, a harassment complaint that was never investigated—can cost millions in regulatory penalties, legal settlements, and reputational harm. These costs are almost never symmetric, and the asymmetry should drive threshold selection.

Third, reviewer capacity needs to be modeled. If the compliance team can review N items per day, the threshold must be set so that the expected daily volume of escalated items is sustainable. Lowering the threshold captures more potential violations but requires proportionally more reviewer capacity.

Fourth, thresholds should be monitored and adjusted on an ongoing basis. As the model improves through feedback, as communication patterns shift, and as regulatory expectations evolve, the optimal threshold will need to move with them.

Alert fatigue is the silent killer of compliance systems. Research in healthcare alarm management has shown that when the ratio of false to true alarms is too high, clinicians begin ignoring critical alerts entirely. The same dynamic applies in compliance. A system in which reviewers encounter 99 false positives for every true positive is arguably *worse* than no automated system at all, because it creates the illusion of monitoring while actively degrading human vigilance (Anand and Sengupta, 2024).

23.3 The Feedback Loop

The human review process generates a valuable byproduct: labeled training data. Every time a reviewer marks a flagged item as a true positive (an actual violation) or a false positive (benign), that judgment—together with the original communication text and the model's risk score—can feed back into the system.

This feedback serves three functions. First, classification and scoring models can be

periodically *retrained* on the accumulated reviewer decisions. This is supervised learning operating in production: the model learns from the very review process it supports, and its accuracy improves with each cycle. Second, for LLM-based scoring, reviewer feedback reveals patterns the model consistently misinterprets. The system prompt can be *refined* to address these specific failure patterns—for instance, adding explicit instructions about idiomatic phrases that are routinely flagged in error. Third, as model performance improves, the optimal *thresholds* may shift: better precision at the same recall level means fewer items need to be escalated, which reduces reviewer burden.

The result is a virtuous cycle. Better models produce fewer false positives, which reduces alert fatigue, which leads to more attentive reviews, which generates higher-quality training data, which produces better models. Organizations that invest in this feedback infrastructure see compounding returns over time. On the other hand, a compliance model that is never updated will gradually erode as language patterns and regulatory expectations shift.

23.4 Explainability and Audit Trails

For a compliance system to earn the trust of stakeholders, accuracy alone is insufficient. The system must also be *transparent*.

Explainable Flagging. When the system flags a communication, it must be able to explain why. This means surfacing the specific phrases or patterns that triggered the flag, the risk category that was detected, the model's confidence level, and the contextual factors that contributed to the score. Without this explanation, reviewers cannot efficiently evaluate flagged items, and auditors cannot assess whether the system is functioning as intended. Explainability is a prerequisite for regulatory acceptance.

Audit Trails. Every decision in the system should be recorded in an audit trail. For each communication assessed, the log should capture the timestamp of the assessment, the model's score and explanation, the reviewer's determination and written rationale (if the item reached human review), and any escalation or follow-up actions taken. This audit trail serves regulatory examination defense, internal compliance reporting, model performance analysis, and legal documentation. In a regulatory examination, the ability to produce a clear record of what was reviewed, by whom, and with what outcome is often as important as the quality of the reviews themselves.

23.5 Why Full Automation Is Insufficient

A natural question is whether improving model accuracy will eventually make human review unnecessary. For the foreseeable future, the answer is no, for several reasons that are structural rather than merely a reflection of current model limitations.

Many regulations explicitly require human oversight in compliance determinations. FINRA Rule 3110 mandates supervisory review of broker-dealer communications. GDPR Article 22 gives individuals the right not to be subject to decisions based solely on automated processing. Various healthcare regulations require human judgment in clinical compliance reviews. A fully automated system may not satisfy these legal requirements regardless of its accuracy.

Beyond regulatory mandates, *accountability* demands human involvement. When a compliance decision is challenged by a regulator, the organization must demonstrate that a qualified person evaluated the evidence and exercised professional judgment. An automated score, no matter how accurate, does not satisfy this standard.

There is also the problem of context that models cannot access. Language is inherently ambiguous, and a message that appears suspicious may be entirely innocent in light of information the model does not have: a prior verbal conversation, an organizational relationship, or an industry convention. Human reviewers bring institutional knowledge

and common sense that models often lack.

Finally, compliance standards are not static. New regulations, enforcement actions, and judicial decisions continuously reshape what constitutes a violation. Human reviewers apply evolving professional judgment; a trained model applies the patterns present in its training data, which may be outdated. And in adversarial contexts—where individuals actively attempt to evade detection—human reviewers can recognize novel evasion tactics that a model was never trained to detect.

23.6 Model Monitoring and Drift

Compliance models can degrade silently, and without active monitoring, an organization may not discover the deterioration until a regulatory examination or a missed violation forces the issue. Three monitoring signals help detect degradation before it causes real harm.

First, the compliance team should track the average risk score the model assigns across all communications over time. If that average suddenly rises or falls without an obvious explanation, something has changed—either the communications themselves are different (a new business line, a shift in employee behavior), the model is malfunctioning, or both. A sudden shift warrants investigation.

Second, when human reviewers increasingly dismiss items the model flagged as high-risk, it suggests the model is no longer well-calibrated to the current mix of communications. This is one of the earliest and most reliable signs that the model needs retraining or prompt revision.

Models are trained on the communication patterns that existed at training time. When the organization adopts a new messaging platform, acquires a company with different communication norms, or enters a new line of business, the model may have no experience with the resulting language patterns. These blind spots must be actively identified by comparing the model's training data against the current communication landscape.

Finally, a good practice is *adversarial testing*: periodically feeding the system synthetic messages that mimic known violation patterns and confirming that it still flags them. This is the compliance equivalent of a fire drill—it is far better to discover that a detection capability has degraded during a controlled test than during a regulatory examination.

References

Anand, Apurva, and Saibal Sengupta. 2024. "Alert Fatigue in Compliance Monitoring: Challenges and Machine-Learning-Based Solutions." *Journal of Financial Compliance*, 7(3). Discusses the alert fatigue problem in compliance systems.

Bank for International Settlements. 2024. "Anomaly Detection in Financial Statements: Bank Al-Maghrib Case Study." Irving Fisher Committee on Central Bank Statistics IFC Bulletin No. 65.

Bao, Jie, et al. 2023. "Applications of Natural Language Processing in Financial Regulation and Compliance: A Survey." *ACM Computing Surveys*. Survey of NLP methods in regulatory compliance.

Bellman, Richard. 1957. *Dynamic Programming*. Princeton University Press.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning*, 24(2): 123–140.

Breiman, Leo. 2001*a*. "Random Forests." *Machine Learning*, 45(1): 5–32.

Breiman, Leo. 2001*b*. "Statistical Modeling: The Two Cultures." *Statistical Science*, 16(3): 199–231.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. "Language Models Are Few-Shot Learners." Vol. 33, 1877–1901. The GPT-3 paper; introduced few-shot prompting.

Bujel, Kenneth, Francis Lai, and Edward Szczerbicki. 2019. "Solving High Volume Capacitated Vehicle Routing Problem with Time Windows Using Recursive-DBSCAN Clustering Algorithm."

Cai, Han, Kan Ren, Weinan Zhang, Kleanthis Malber, Jun Wang, Yong Yu, and Tie-Yan Liu. 2018. "Real-Time Bidding by Reinforcement Learning in Display Advertising." arXiv:1803.00259.

Campello, Ricardo J. G. B., Davoud Moulavi, and Jörg Sander. 2013. "Density-Based Clustering Based on Hierarchical Density Estimates." *Lecture Notes in Computer Science*, 7819: 160–172.

Chawla, Nitesh V., Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. "SMOTE: Synthetic Minority Over-Sampling Technique." *Journal of Artificial Intelligence Research*, 16: 321–357.

Chen, Shi-Yong, Yang Yu, Qing Da, Jun Tan, Hai-Kuan Huang, and Hai-Hong Tang. 2018. "Stabilizing Reinforcement Learning in Dynamic Environment with Application to Online Recommendation." 1187–1196.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." 785–794.

Christiano, Paul F., Jan Leike, Tom B. Brown, Miljan Marber, Shane Legg, and Dariusz Amodei. 2017. "Deep Reinforcement Learning from Human Preferences." Vol. 30. Foundational paper on RLHF.

Christophides, Vassilis, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2021. "An Overview of End-to-End Entity Resolution for Big Data." Vol. 53, 1–42. Comprehensive survey of entity resolution methods.

Cormack, Gordon V., Charles L. A. Clarke, and Stefan Buettcher. 2009. "Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods." 758–759.

Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." *Machine Learning*, 20(3): 273–297.

Cover, Thomas, and Peter Hart. 1967. "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory*, 13(1): 21–27.

Cybenko, George. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals, and Systems*, 2(4): 303–314. Universal approximation theorem.

Databricks Labs. 2021. "Identifying Financial Fraud with Geospatial Clustering." *https://www.databricks.com/blog/2021/04/13/identifying-financial-fraud-with-geospatial-clustering.html*.

DeepMind. 2016. "DeepMind AI Reduces Google Data Centre Cooling Bill by 40%." *https://deepmind.google/discover/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-by-40/*.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." 4171–4186.

DoorDash. 2025. "2025 Summer Intern Projects, Part 2." *https://careersatdoordash.com/blog/part-2-doordash-2025-summer-intern-projects/*.

Dudík, Miroslav, Dumitru Erhan, John Langford, and Lihong Li. 2014. "Doubly Robust Policy Evaluation and Optimization." *Statistical Science*, 29(4): 485–511.

Duhigg, Charles. 2012. "How Companies Learn Your Secrets." *The New York Times Magazine*.

Es, Shahul, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. 2024. "RAGAS: Automated Evaluation of Retrieval Augmented Generation." *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (EACL): System Demonstrations*. Framework for evaluating RAG systems on faithfulness, relevance, and context quality.

Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." 226–231.

Fawcett, Tom. 2006. "An Introduction to ROC Analysis." *Pattern Recognition Letters*, 27(8): 861–874.

Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. 2019. "All Models Are Wrong, but Many Are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously." *Journal of Machine Learning Research*, 20(177): 1–81.

Freund, Yoav, and Robert E. Schapire. 1997. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting." *Journal of Computer and System Sciences*, 55(1): 119–139.

Friedman, Jerome H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *Annals of Statistics*, 29(5): 1189–1232.

Gao, Luyu, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2023. "Precise Zero-Shot Dense Retrieval Without Relevance Labels." *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*. Introduces the HyDE (Hypothetical Document Embeddings) technique.

Gao, Yunfan, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. "Retrieval-Augmented Generation for Large Language Models: A Survey." *arXiv preprint arXiv:2312.10997*.

Geman, Stuart, Elie Bienenstock, and René Doursat. 1992. "Neural Networks and the Bias/Variance Dilemma." *Neural Computation*, 4(1): 1–58.

GitHub Customer Success. 2024. "AI-Powered Customer Feedback Analysis at Scale." *https://www.zenml.io/llmops-database/ai-powered-customer-feedback-analysis-at-scale*.

GOGOX Technology. 2022. "Improving Operations with Route Optimization." *https://medium.com/gogox-technology/improving-operations-with-route-optimization-e032d8bf5edc*.

Goldstein, Alex, Adam Kapelner, Justin Bleich, and Emil Pitkin. 2015. "Peeking Inside the Black Box: Visualizing Statistical Learning with Plots of Individual Conditional Expectation." *Journal of Computational and Graphical Statistics*, 24(1): 44–65.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press.

Grab Engineering. 2018. "The Data and Science Behind GrabShare, Part I." *https://engineering.grab.com/the-data-and-science-behind-grabshare-part-i*.

Grootendorst, Maarten. 2022. "BERTopic: Neural Topic Modeling with a Class-Based TF-IDF Procedure." *arXiv preprint arXiv:2203.05794*. Combines sentence embeddings, UMAP, and HDBSCAN for topic modeling.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* . 2nd ed., Springer. Commonly known as ESL.

Hill, Kashmir. 2012. "How Target Figured Out a Teen Girl Was Pregnant Before Her Father Did." *Forbes*.

Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. 2006. "Reducing the Dimensionality of Data with Neural Networks." *Science*, 313(5786): 504–507.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation*, 9(8): 1735–1780.

Hoerl, Arthur E., and Robert W. Kennard. 1970. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics*, 12(1): 55–67.

Huang, Lei, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. "A

Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions." *arXiv preprint arXiv:2311.05232*.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2023. *An Introduction to Statistical Learning: With Applications in Python.* . 2nd ed., Springer. Commonly known as ISLR.

Järvelin, Kalervo, and Jaana Kekäläinen. 2002. "Cumulated Gain-Based Evaluation of IR Techniques." *ACM Transactions on Information Systems*, 20(4): 422–446. Introduces the NDCG metric.

Johnson, Jeff, Matthijs Douze, and Hervé Jégou. 2021. "Billion-Scale Similarity Search with GPUs." *IEEE Transactions on Big Data*, 7(3): 535–547. Describes FAISS, the foundational library for approximate nearest neighbor search.

Jolliffe, Ian T., and Jorge Cadima. 2016. "Principal Component Analysis: A Review and Recent Developments." *Philosophical Transactions of the Royal Society A*, 374(2065).

JPMorgan. 2017. "LOXM: AI-Powered Execution System." *https://www.marketsmedia.com/jpmorgan-aims-to-expand-ai-tool-loxm/*.

Jurafsky, Daniel, and James H. Martin. 2024. *Speech and Language Processing.* . 3rd (draft) ed. Freely available at https://web.stanford.edu/~jurafsky/slp3/. The standard NLP textbook.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dariusz Amodei. 2020. "Scaling Laws for Neural Language Models." *arXiv preprint arXiv:2001.08361*.

Karpukhin, Vladimir, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. "Dense Passage Retrieval for Open-

Domain Question Answering." 6769–6781. Foundational paper on dense (embedding-based) retrieval.

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." Vol. 30.

Kendall, Alex, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Mayol-Cuevas, and Nicolai Birkbeck. 2019. "Learning to Drive in a Day." *arXiv preprint arXiv:1807.00412*. Wayve.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." Vol. 25.

Kuhn, Max, and Kjell Johnson. 2019. "Feature Engineering and Selection: A Practical Approach for Predictive Models."

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE*, 86(11): 2278–2324.

Lewis, Patrick, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." Vol. 33, 9459–9474. The paper that introduced the RAG framework.

Liu, Nelson F., Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. "Lost in the Middle: How Language Models Use Long Contexts." *Transactions of the Association for Computational Linguistics*, 12: 157–173.

Lloyd, Stuart P. 1982. "Least Squares Quantization in PCM." *IEEE Transactions on Information Theory*, 28(2): 129–137. Original K-means algorithm.

McInnes, Leland, John Healy, and James Melville. 2018. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction." *arXiv preprint arXiv:1802.03426*.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." Introduces Word2Vec.

Milliman. 2025. "AI-Supported Anomaly Detection in Insurance."

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature*, 518(7540): 529–533.

Molnar, Christoph. 2022. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable.* . 2nd ed.

Nogueira, Rodrigo, and Kyunghyun Cho. 2019. "Passage Re-Ranking with BERT." Seminal paper on neural reranking for information retrieval.

OpenAI. 2019. "Solving Rubik's Cube with a Robot Hand." *https://openai.com/index/solving-rubiks-cube/*.

Ouyang, Long, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." *Advances in Neural Information Processing Systems*, 35. The InstructGPT paper; describes the RLHF pipeline used to align GPT models.

Prokhorenkova, Liudmila, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. "CatBoost: Unbiased Boosting with Categorical Features." *Advances in Neural Information Processing Systems*, 31.

Quinlan, J. Ross. 1986. "Induction of Decision Trees." *Machine Learning*, 1(1): 81–106.

Reimers, Nils, and Iryna Gurevych. 2019. "Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks." 3982–3992. Foundational paper for sentence-level embeddings used in semantic search.

Rousseeuw, Peter J. 1987. "Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis." *Journal of Computational and Applied Mathematics*, 20: 53–65.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature*, 323(6088): 533–536.

Sahoo, Pranab, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. "A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications." *arXiv preprint arXiv:2402.07927*.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." 1715–1725. Introduces Byte-Pair Encoding (BPE) for tokenization.

Shapley, Lloyd S. 1953. "A Value for n-Person Games." In *Contributions to the Theory of Games*. Vol. II, , ed. Harold W. Kuhn and Albert W. Tucker, 307–317. Princeton University Press.

Shi, Hao-Ran, et al. 2019. "Virtual-Taobao: Virtualizing Real-World Online Retail Environment for RL-Based Recommendation and Inventory Management." arXiv:1912.02572. Describes both RL pricing and inventory experiments in the Alibaba ecosystem.

Stone, Mervyn. 1974. "Cross-Validatory Choice and Assessment of Statistical Predictions." *Journal of the Royal Statistical Society: Series B*, 36(2): 111–133.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* . 2nd ed., MIT Press.

Tibshirani, Robert. 1996. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society: Series B*, 58(1): 267–288.

van der Maaten, Laurens, and Geoffrey Hinton. 2008. "Visualizing Data Using t-SNE." *Journal of Machine Learning Research*, 9(86): 2579–2605.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." Vol. 30.

Wachter, Sandra, Brent Mittelstadt, and Chris Russell. 2018. "Counterfactual Explanations Without Opening the Black Box: Automated Decisions and the GDPR." *Harvard Journal of Law and Technology*, 31(2): 841–887.

Watkins, Christopher J. C. H., and Peter Dayan. 1992. "Q-Learning." *Machine Learning*, 8(3–4): 279–292.

Weinberger, Kilian Q., and Lawrence K. Saul. 2009. "Distance Metric Learning for Large Margin Nearest Neighbor Classification." Vol. 10, 207–244.

Wei, Xiang, Xingyu Cui, Ning Cheng, Xiaobin Wang, Xin Zhang, Shen Huang, Pengjun Xie, Jinan Xu, Yufeng Chen, Meishan Zhang, Yong Jiang, and Wenjuan Han. 2023. "Zero-Shot Information Extraction via Chatting with ChatGPT." *arXiv preprint arXiv:2302.10205*.

Zhang, Yuwei, Zihan Zhan, Shivam Jain, Hidetaka Kamigaito, and Manabu Okumura. 2023. "ClusterLLM: Large Language Models as a Guide for Text Clustering."

Zhao, Wayne Xin, Kun Zhou, Junyi Li, Tieyun Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. "A Survey of Large Language Models." *arXiv preprint arXiv:2303.18223*. Comprehensive survey of LLM architectures, training, and capabilities.