13 Machine Learning

R. Heller · Date: 2026-05-11 · Last changed: 2026-10-30

Cross-validation, regularisation (ridge/LASSO/elastic net), trees and ensembles, neural networks via `torch` / `keras`, SHAP and other interpretability tools, and the `tidymodels` workflow. False-discovery and knockoff-based selection are treated alongside CV as competing answers to the same question.

This chapter contains **46 method pages** and **8 labs**. If you are not sure which method to read, return to Chapter 0 and follow the decision tree to the right node.

13.1 Method pages

Method	Source slug
A Complete mlr3 Workflow	`mlr3-workflow`
A Complete tidymodels Workflow	`tidymodels-workflow`
Anomaly Detection	`anomaly-detection`
Bagging	`bagging`
Calibration Plots	`calibration-plots`
CatBoost	`catboost`
Class Weights for Imbalanced Data	`class-weights-imbalanced`
Convolutional Neural Networks: Introduction	`cnn-introduction`
Cross-Validation	`cross-validation`
DBSCAN as an ML Tool	`clustering-dbscan-ml`
Decision Trees (CART)	`decision-trees-cart`
Dropout and Early Stopping	`dropout-early-stopping`
Extremely Randomised Trees	`extremely-randomized-trees`
Feature Engineering	`feature-engineering`
Feature Selection	`feature-selection`
Feedforward Networks in torch	`feedforward-networks-torch`
Gradient Boosting	`gradient-boosting`
Isolation Forest	`isolation-forest`
Isotonic Regression Calibration	`isotonic-regression`
k-Means as an ML Tool	`clustering-kmeans-ml`
k-Nearest Neighbours Classification	`knn-classification`
k-Nearest Neighbours Regression	`knn-regression`
Kernel SVMs	`svm-kernel`
LightGBM	`lightgbm`

Method	Source slug
LIME Explanations	`lime-explanations`
Linear Discriminant as ML	`linear-discriminant-ml`
Linear Support Vector Machines	`svm-linear`
Logistic Regression as ML	`logistic-regression-ml`
Naive Bayes	`naive-bayes`
Nested Cross-Validation	`nested-cv`
Neural Networks: Introduction	`neural-networks-intro`
Partial Dependence Plots	`partial-dependence-plots`
Platt Scaling	`platt-scaling`
Preprocessing with recipes	`recipes-preprocessing`
Random Forests	`random-forest`
Regularisation in ML	`regularization-ridge-lasso-ml`
RNNs and LSTMs: Introduction	`rnn-lstm-intro`
SHAP Values	`shap-values`
SMOTE for Imbalanced Classes	`smote-imbalanced`
Stacking Ensembles	`stacking-ensembles`
Supervised Learning: Overview	`supervised-learning-overview`
The Bias-Variance Tradeoff	`bias-variance-tradeoff`
Train-Test Splits	`train-test-split`
Transformers: Overview	`transformer-overview`
Variable Importance	`variable-importance`
XGBoost	`xgboost`

13.2 Labs

Lab
Cross-validation, nested CV, bootstrap .632+
Regularisation: ridge, lasso, elastic net
Trees, random forests, gradient boosting
Interpretability and SHAP
Tabular neural networks with torch
Imaging and sequence models intro
tidymodels pipelines
FDR, knockoffs, replication crisis

13.3 Anomaly Detection

13.3.1 Introduction

Anomaly detection flags observations that deviate from typical patterns. It is used for fraud detection, sensor fault monitoring, network intrusion, and adverse-event surveillance. Unlike classification, labels for anomalies are often unavailable at training time – methods are unsupervised or semi-supervised.

13.3.2 Prerequisites

Multivariate data; distance / density concepts.

13.3.3 Theory

Distance-based: kNN distance or average distance to neighbours; high = anomaly.

Density-based: LOF (Local Outlier Factor) compares local density to neighbours.

One-class SVM: learns a boundary enclosing most of the training data; points outside are anomalies.

Isolation Forest: exploits that anomalies are easier to isolate in random trees (shorter path length to terminal node).

Autoencoders: train a reconstruction network on normal data; anomalies have high reconstruction error.

13.3.4 Assumptions

Training data is mostly “normal”; anomaly pattern differs in some quantifiable way from normal.

13.3.5 R Implementation

```r
library(e1071)

set.seed(2026)
n <- 300
# Normal data is a ball; add some outliers
df <- data.frame(
  x1 = c(rnorm(n, 0, 1), runif(20, -5, 5)),
  x2 = c(rnorm(n, 0, 1), runif(20, -5, 5))
)
labels <- c(rep("normal", n), rep("anomaly", 20))

# One-class SVM
oc <- svm(df, type = "one-classification", nu = 0.1, kernel = "radial")
pred <- predict(oc, df)                # TRUE = inlier, FALSE = anomaly

table(truth = labels, flagged = ifelse(pred, "inlier", "anomaly"))
```

13.3.6 Output & Results

Confusion matrix of the one-class SVM against ground truth; it correctly flags most of the 20 injected anomalies as outliers.

13.3.7 Interpretation

“One-class SVM detected 17 of 20 injected anomalies (sensitivity 0.85) with a 6 % false-positive rate on normal data. Isolation Forest gave similar results with less tuning.”

13.3.8 Practical Tips

- Start with Isolation Forest; it is fast, robust, and has few hyperparameters.
- `nu` in one-class SVM approximates the expected anomaly fraction.
- Autoencoders are competitive for complex high-dimensional anomalies (images, signals).
- Validate with a held-out labelled anomaly set; unsupervised evaluation is hard.
- Combine multiple detectors for robustness; anomalies are rare and single-method false-positive rates compound.

13.3.9 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.3.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.4 Bagging

13.4.1 Introduction

Bagging — bootstrap aggregation, introduced by Breiman in 1996 — trains B models on B bootstrap samples of the training data and averages their predictions. The averaging dramatically reduces variance for high-variance, low-bias learners (deep decision trees,

kNN with small k) without increasing their bias. Random forests are bagged trees with an additional feature-randomisation step that decorrelates the individual trees and amplifies the variance reduction.

13.4.2 Prerequisites

A working understanding of bootstrap resampling, the bias-variance trade-off, and decision trees as the canonical high-variance base learner.

13.4.3 Theory

For B predictions $\hat{y}_b$ each with variance σ^2 and pairwise correlation ρ, the variance of their mean is

$$\operatorname{Var}\left(\frac{1}{B}\sum_{b}\hat{y}_b\right) = \rho\sigma^2 + \frac{(1-\rho)\sigma^2}{B}.$$

As B grows, the second term vanishes; the first term is the residual variance limited by correlation. Bagging alone reduces variance via the $(1-\rho)/B$ contribution; adding decorrelation (feature subsampling, as in random forest) lowers ρ and produces further variance reduction.

Out-of-bag (OOB) observations — the roughly 37 % of rows not selected by each bootstrap — provide a free held-out validation set: for each row, average predictions from trees that did not include it during training.

13.4.4 Assumptions

The base learner benefits from variance reduction; bagging is most effective on high-variance learners (deep trees, kNN with small k, neural networks) and does essentially nothing for stable learners like linear regression.

13.4.5 R Implementation

```r
library(ipred); library(rpart)

set.seed(2026)
n <- 400
df <- data.frame(
  x1 = rnorm(n), x2 = rnorm(n),
  y  = factor(ifelse(plogis(1 * rnorm(n) + 0.5 * rnorm(n)^2) > runif(n),
                     "yes", "no"))
)

# Bagged trees (no feature subsampling -- contrast with random forest)
m <- bagging(y ~ ., data = df, nbagg = 100, coob = TRUE)
print(m)
```

13.4.6 Output & Results

`ipred::bagging()` returns a fitted bagged ensemble with the OOB misclassification rate when `coob = TRUE`. The OOB error is the standard generalisation estimate; comparing it to a single tree's CV error quantifies the variance reduction from bagging.

13.4.7 Interpretation

A reporting sentence: "100 bagged trees achieved OOB misclassification rate of 0.21, an improvement over a single tree's CV error of 0.28; switching to a random forest with feature subsampling reduced OOB further to 0.18, illustrating that decorrelation contributes additional variance reduction beyond plain bagging." Always compare bagged ensembles against single-base-learner baselines.

13.4.8 Practical Tips

- Bagging primarily benefits high-variance learners (trees, kNN with small k); linear models are stable and gain little.
- Use $B = 100$ to 500 bootstrap replicates; returns diminish quickly beyond that, and computation grows linearly.
- OOB error is nearly unbiased for moderate to large n and free to compute alongside training; prefer it to cross-validation for rapid iteration.
- Random forests (bagging + feature subsampling) almost always outperform plain bagged trees; reach for `ranger` directly rather than tuning `ipred::bagging`.
- For classification, average class probabilities (soft voting) rather than class labels (hard voting); soft voting is consistently more accurate.
- Bagging extends beyond trees: any learner can be bagged via `ipred::bagging` with `bf` set to a custom training function.

13.4.9 R Packages Used

`ipred::bagging()` for canonical bagging on arbitrary base learners; `ranger` and `randomForest` for bagged trees with feature subsampling (random forests); `parsnip::bag_tree()` for tidymodels integration; `mlr3pipelines::po("subsample")` for bagging within mlr3 pipelines.

13.4.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.4.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+

- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.5 The Bias-Variance Tradeoff

13.5.1 Introduction

Every supervised-learning error decomposes into structural bias (the wrong model class cannot fit the truth), variance (sampling noise in the fit), and irreducible noise. The bias-variance tradeoff is the single most important mental model for picking between a rigid and a flexible estimator.

13.5.2 Prerequisites

Regression basics; expected value and variance of estimators.

13.5.3 Theory

For squared loss and a target $f(x) + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$, $\mathrm{Var}(\varepsilon) = \sigma^2$:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{(\mathbb{E}[\hat{f}(x)] - f(x))^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2]}_{\text{Variance}} + \sigma^2.$$

Complex models: low bias, high variance. Simple models: high bias, low variance. Regularisation, averaging (bagging), and dropout explicitly shift along this curve.

13.5.4 Assumptions

Squared loss and iid samples; trained estimator is a random function of the training set.

13.5.5 R Implementation

```r
set.seed(2026)

true_f ← function(x) sin(x)
x_test ← seq(0, 2 * pi, length.out = 100)
f_true ← true_f(x_test)

n_reps  ← 200; n_train ← 30

simulate_fit ← function(degree) {
  preds ← replicate(n_reps, {
    x_train ← runif(n_train, 0, 2 * pi)
    y_train ← true_f(x_train) + rnorm(n_train, 0, 0.3)
    mdl ← lm(y_train ~ poly(x_train, degree))
    predict(mdl, newdata = data.frame(x_train = x_test))
  })
  bias ← rowMeans(preds) - f_true
  variance ← apply(preds, 1, var)
  list(bias2 = mean(bias^2), variance = mean(variance))
}

sapply(c(1, 3, 7, 15), simulate_fit)
```

13.5.6 Output & Results

As polynomial degree grows, bias^2 falls and variance rises; the sum (excess MSE beyond noise) is minimised at an intermediate flexibility.

13.5.7 Interpretation

“Polynomial degree 3 balanced bias (0.12) and variance (0.09), minimising excess MSE; degree 15 reduced bias to 0.001 but inflated variance to 0.45, hurting generalisation.”

13.5.8 Practical Tips

- Overfitting corresponds to high variance (the same structure fits differently across training resamples).
- Underfitting corresponds to high bias (structurally too restrictive to capture the signal).
- Ensembling (bagging, random forests) reduces variance without increasing bias.
- Regularisation (ridge, lasso) increases bias but reduces variance – tune by CV.
- Deep learning’s “double descent” is a modern complication; the classical curve is the best starting intuition.

13.5.9 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.5.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.6 Calibration Plots

13.6.1 Introduction

A calibration plot (reliability diagram) plots predicted probability against observed frequency. A well-calibrated classifier's curve lies on the diagonal: when it predicts 0.7, events occur 70 % of the time. Poor calibration means probabilities cannot be trusted for decision thresholds or risk communication.

13.6.2 Prerequisites

Classification; predicted probabilities; binning.

13.6.3 Theory

Bin predictions by predicted probability; compute the observed frequency in each bin. Plot bin midpoints vs observed frequencies. A perfect classifier follows $y = x$; curves below indicate over-confidence, curves above indicate under-confidence.

Summary metrics: **Brier score** $= \mathbb{E}[(p - y)^2]$, **Expected Calibration Error (ECE)** $= \sum_k w_k |\mathrm{acc}_k - \mathrm{conf}_k|$.

13.6.4 Assumptions

Predictions are valid probabilities in $[0, 1]$; sufficient data per bin (10-20 bins for $n > 1000$).

13.6.5 R Implementation

```r
library(probably); library(yardstick); library(dplyr); library(tibble);
library(ggplot2)

set.seed(2026)
n ← 1000
# Simulate mis-calibrated predictions
true_y ← rbinom(n, 1, 0.4)
raw_p  ← plogis(-1 + 2 * true_y + rnorm(n, 0, 1.5))
# Inflate confidence to produce over-confident predictions
p_overconf ← raw_p^0.6
p_overconf ← pmin(pmax(p_overconf, 0.02), 0.98)

df ← tibble(truth = factor(true_y, levels = c(1, 0)),
            pred  = p_overconf)

cal_df ← cal_plot_breaks(df, truth, pred, num_breaks = 10)

# Manual reliability data + plot
df$bin ← cut(df$pred, seq(0, 1, 0.1), include.lowest = TRUE)
reliability ← df %>%
  group_by(bin) %>%
  summarise(mean_pred = mean(pred),
            obs_freq  = mean(as.integer(truth == "1")),
            n = n())

ggplot(reliability, aes(mean_pred, obs_freq)) +
  geom_point(size = 2, colour = "#2A9D8F") +
  geom_line() +
  geom_abline(linetype = 2) +
  labs(x = "Predicted probability", y = "Observed frequency",
       title = "Reliability diagram (simulated over-confident model)") +
  theme_minimal()

mean((df$pred - as.integer(df$truth == "1"))^2)   # Brier score
```

13.6.6 Output & Results

A reliability diagram (scatter of bin-means) and the Brier score. The simulated over-confident model lies above the diagonal at low predicted probabilities.

13.6.7 Interpretation

“The classifier over-predicts at low probabilities and under-predicts at high probabilities; Brier score 0.22 indicates moderate calibration error. Platt or isotonic recalibration on a validation set would flatten the curve toward the diagonal.”

13.6.8 Practical Tips

- Tree ensembles (RF, XGBoost) are usually mis-calibrated; apply Platt scaling or isotonic regression on a validation set.
- Neural networks with modern training (high-capacity, low label smoothing) often over-confident; temperature scaling fixes it.
- Use stratified bins for reliable plots when probabilities concentrate near 0 or 1.

- Report Brier score and ECE alongside ROC AUC; AUC is rank-based and does not reveal calibration.
- For clinical applications, calibration matters more than discrimination; always verify on deployment-era data.

13.6.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.6.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.7 CatBoost

13.7.1 Introduction

CatBoost (Prokhorenkova et al., 2018) focuses on principled handling of categorical features via ordered target encoding – computing target statistics only from prior rows to avoid target leakage. It also uses **symmetric (oblivious) trees**: trees where every split at a given depth uses the same feature and threshold, yielding fast inference and natural regularisation.

13.7.2 Prerequisites

Gradient boosting; categorical encoding basics.

13.7.3 Theory

Ordered target encoding: for a categorical feature, the encoding of row i uses target statistics computed only over rows 1..i-1 (under a random permutation). This prevents target leakage present in ordinary mean-target encoding.

Ordered boosting: each tree is trained on residuals computed using an independent prefix of data, reducing prediction-shift bias.

Symmetric trees: fixed structure across layers; inference reduces to look-up tables, making CatBoost extremely fast at inference time.

13.7.4 Assumptions

Categorical features are correctly flagged; data are reasonably uniform (no temporal drift within a training set).

13.7.5 R Implementation

```
# library(catboost)  # install from https://catboost.ai/en/docs/installation/r-installation-binary-installation

set.seed(2026)
n ← 500
df ← data.frame(
  cat1 = sample(letters[1:5], n, replace = TRUE),
  cat2 = sample(c("a", "b", "c"), n, replace = TRUE),
  num1 = rnorm(n), num2 = rnorm(n),
  y    = factor(sample(c("0", "1"), n, replace = TRUE))
)

# pool ← catboost.load_pool(
#   data = df[, -5], label = as.integer(df$y) - 1,
#   cat_features = c(0, 1))
#
# model ← catboost.train(
#   learn_pool = pool,
#   params = list(iterations = 500, learning_rate = 0.05,
#                 depth = 6, loss_function = "Logloss",
#                 verbose = 0))
#
# importance ← catboost.get_feature_importance(model)
# importance

# Run summary for illustration (the fit itself needs the package install)
head(df)
```

13.7.6 Output & Results

A trained CatBoost model with feature-importance scores; categorical features are ordered-target-encoded internally.

13.7.7 Interpretation

“CatBoost achieved log-loss 0.36 on the held-out set. Feature importance ranked the categorical ‘cat1’ highest, leveraging ordered target encoding to extract signal that plain one-hot encoding misses.”

13.7.8 Practical Tips

- Pass categorical columns as factor or integer; flag via `cat_features`.
- CatBoost handles missing values natively; no imputation needed.
- Symmetric trees are fast at inference; useful when deployment latency matters.

- Default hyperparameters are already strong; tune learning rate and depth first.
- On small data, CatBoost generally outperforms XGBoost/LightGBM on heavily categorical datasets.

13.7.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.7.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.8 Class Weights for Imbalanced Data

13.8.1 Introduction

Class weighting addresses imbalanced classification by scaling each observation's loss contribution inversely to its class frequency, giving the minority class more influence on the gradient or split decision. It achieves similar effects to SMOTE-style oversampling but without synthesising new data: simpler, deterministic, easier to reproduce, and natively supported by most classifiers (logistic regression, random forest, XGBoost, neural networks). For most imbalanced problems, class weights are the right starting point before reaching for resampling.

13.8.2 Prerequisites

A working understanding of classification loss functions, the accuracy paradox in imbalanced classification, and basic familiarity with resampling alternatives (oversampling, undersampling, SMOTE).

13.8.3 Theory

The weighted loss is

$$L = \sum_i w_{y_i} \cdot \ell\big(f(x_i), y_i\big),$$

where w_k is the weight for class k. Typical balanced weights use the inverse class frequency: $w_k = n/(K n_k)$ where K is the number of classes and n_k is the count of class k. This makes the total weight per class equal, so each class contributes equally to the loss regardless of its sample size.

In tree-based methods, class weights enter the split-impurity computation; in gradient-boosted methods, they enter the gradient and Hessian; in neural networks, they multiply the per-example loss before summing.

13.8.4 Assumptions

The problem is genuinely a classification task with imbalanced classes; the imbalance reflects the population (not just an artefact of training-data collection that will not transfer to deployment).

13.8.5 R Implementation

```r
library(ranger)

set.seed(2026)
n <- 500
df <- data.frame(
  x1 = rnorm(n), x2 = rnorm(n),
  y  = factor(sample(c("pos", "neg"), n, replace = TRUE,
                     prob = c(0.05, 0.95)))
)

idx <- sample(n, n * 0.7)
train <- df[idx, ]; test <- df[-idx, ]

rf_unw <- ranger(y ~ ., data = train, num.trees = 200,
                 classification = TRUE)
pred_unw <- predict(rf_unw, test)$predictions

# Inverse-frequency class weights
cw <- setNames(as.numeric(1 / prop.table(table(train$y))),
               names(table(train$y)))
rf_w <- ranger(y ~ ., data = train, num.trees = 200,
               classification = TRUE, class.weights = cw)
pred_w <- predict(rf_w, test)$predictions

table(truth = test$y, unweighted = pred_unw)
table(truth = test$y, weighted   = pred_w)
```

13.8.6 Output & Results

Two confusion matrices: the unweighted random forest predicts almost exclusively the majority class (the rational behaviour given 95:5 imbalance and accuracy-maximising training); the weighted RF recovers more minority-class examples at the cost of a few additional false positives.

13.8.7 Interpretation

A reporting sentence: “Inverse-frequency class weighting on the 95:5 imbalanced dataset improved minority-class recall from 0.12 to 0.56 with a modest decrease in majority precision (0.97 to 0.93); the weighted classifier is more useful for screening applications where missing minority cases is more costly than false positives.” Always state the weighting scheme and the trade-off in terms of the application’s cost structure.

13.8.8 Practical Tips

- Always try class weights before SMOTE-style oversampling; they are deterministic, faster, and produce reproducible results.
- Tune weights by cross-validation; inverse-frequency is a sensible default but not necessarily optimal for the cost structure of every application.
- All major classifiers support class weights natively: `glmnet(weights = ...)`, `ranger(class.weights = ...)`, `xgboost(scale_pos_weight = ...)`, `keras::compile(... loss_weights = ...)`.
- Weighting shifts the decision boundary; inspect calibration plots and ROC curves to check probabilistic quality after weighting.
- For severely imbalanced problems (below 1 %), combine class weights with threshold tuning on a validation set; the default 0.5 cutoff is rarely optimal under heavy imbalance.
- For problems where the cost of errors is asymmetric and known, build a cost-sensitive classifier directly (`mlr3::msr("classif.costs")`) rather than tuning weights heuristically.

13.8.9 R Packages Used

`ranger` for random forest with native class weights; `glmnet` for penalised logistic regression with `weights = ...`; `xgboost::xgboost(scale_pos_weight = ...)` for gradient-boosted trees; `themis` for SMOTE-style alternatives within tidymodels; `parsnip::set_args(class_weights = ...)` for class-weight support across multiple engines.

13.8.10 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.8.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting

- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.9 DBSCAN as an ML Tool

13.9.1 Introduction

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) groups points that are densely packed together and flags isolated points as noise. Unlike k-means it does not require specifying the number of clusters, handles non-convex cluster shapes, and produces an outlier-detection signal as a side effect — every point flagged as noise is a candidate anomaly. For machine-learning workflows where natural cluster discovery and outlier flagging are both useful, DBSCAN is the right starting point.

13.9.2 Prerequisites

A working understanding of distance metrics, density-based reasoning, and the limitations of partitioning algorithms (k-means) for non-convex cluster shapes.

13.9.3 Theory

DBSCAN has two hyperparameters: `eps` (the neighbourhood radius) and `minPts` (the minimum number of points required for a dense region). Three categories of point emerge from the algorithm:

- **Core point**: at least `minPts` neighbours within `eps`.
- **Border point**: within `eps` of a core point but not itself a core point.
- **Noise point**: neither core nor border.

Clusters are formed as maximally connected sets of core points (with their associated border points). Choosing `eps` is the practical challenge; the standard heuristic is the k-distance plot, which shows the distance to the k-th nearest neighbour for each point sorted ascending — a knee in the curve marks a sensible `eps`.

13.9.4 Assumptions

Cluster density is roughly uniform; the chosen `eps` matches the natural scale of the data; Euclidean distance is meaningful (or another distance has been substituted).

13.9.5 R Implementation

```r
library(dbscan)

set.seed(2026)
n <- 200
x1 <- c(rnorm(n/2, -2, 0.3), rnorm(n/2, 2, 0.3))
x2 <- c(rnorm(n/2, -2, 0.3), rnorm(n/2, 2, 0.3))
outliers_x1 <- runif(10, -4, 4)
outliers_x2 <- runif(10, -4, 4)
df <- data.frame(x1 = c(x1, outliers_x1),
                 x2 = c(x2, outliers_x2))

# Choose eps via the k-distance plot knee
kNNdistplot(df, k = 5); abline(h = 0.5, lty = 2, col = "red")

db <- dbscan(df, eps = 0.5, minPts = 5)
table(cluster = db$cluster)

plot(df, col = db$cluster + 1, pch = 16, asp = 1,
     main = "DBSCAN clustering (cluster 0 = noise)",
     xlab = "x1", ylab = "x2")
```

13.9.6 Output & Results

`dbscan()` returns a vector of cluster labels with `0` indicating noise. The k-distance plot guides `eps` selection; the cluster scatter shows densely-packed groups separated by noise points.

13.9.7 Interpretation

A reporting sentence: "DBSCAN with `eps = 0.5` and `minPts = 5` recovered the two natural clusters injected in the simulation (96 of 100 cluster-1 points and 98 of 100 cluster-2 points correctly assigned) and flagged 10 of the 10 injected outliers as noise (cluster 0); the remaining noise calls were 6 boundary points where the local density is genuinely lower." Always state `eps`, `minPts`, and the noise count.

13.9.8 Practical Tips

- Set `eps` via the k-distance plot's knee; it is the most sensitive hyperparameter and a poor choice fails the algorithm completely.
- `minPts = 2 d$ (dimensions) is a sensible starting heuristic; inspect cluster counts and adjust if they are unreasonably few or many.
- For varying-density clusters, switch to HDBSCAN (`dbscan::hdbscan`); it adapts the density threshold per region and is more robust.
- Use the noise label (`cluster == 0`) as a built-in outlier detector; for anomaly-detection tasks, this is often the simplest and most interpretable approach.
- Scale features before clustering (`scale()` or `recipes::step_normalize()`); DBSCAN's Euclidean distance is sensitive to feature scales.
- For high-dimensional data, distance concentration weakens DBSCAN; reduce dimensionality with PCA or UMAP first.

13.9.9 R Packages Used

`dbscan` for `dbscan()`, `hdbscan()`, `optics()`, and the k-distance heuristic; `fpc::dbscan()` for an alternative implementation; `cluster` for related clustering algorithms; `factoextra::fviz_cluster()` for ggplot-style visualisation of DBSCAN output.

13.9.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.9.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.10 k-Means as an ML Tool

13.10.1 Introduction

k-means is usually taught as a stand-alone unsupervised clustering method, but in ML pipelines it is also a versatile feature-engineering primitive. Pre-clustering observations and using cluster membership (or distance-to-centroid) as new features can substantially improve downstream supervised models, especially when the underlying signal is non-linear and a clustering captures local structure that linear models miss. k-means is also the workhorse for vector quantisation, image segmentation, and customer-segmentation tasks.

13.10.2 Prerequisites

A working understanding of Euclidean distances, centroids, the within-cluster sum-of-squares objective, and the bias-variance trade-off in feature engineering.

13.10.3 Theory

k-means minimises the within-cluster sum of squares (WCSS):

$$\min_{C} \sum_{k=1}^{K} \sum_{i \in C_k} \|x_i - \mu_k\|^2.$$

Lloyd's algorithm alternates assignment (each point to its nearest centroid) and update (each centroid to the mean of its assigned points) until convergence. The algorithm is non-convex; multiple random starts are needed to find a good local optimum. The number of clusters K is a hyperparameter chosen by the elbow, silhouette, or gap statistic methods.

For feature engineering, two derived quantities from a k-means fit are useful: - **Cluster membership** (one-hot or label): a categorical feature indicating which cluster each observation belongs to. - **Distance to each centroid**: K continuous features that often carry more predictive information than the cluster label alone.

13.10.4 Assumptions

Clusters are roughly spherical and equal-sized (the implicit Voronoi-cell geometry); features are scaled to comparable units; the chosen K matches a meaningful structure in the data.

13.10.5 R Implementation

```r
set.seed(2026)
n ← 300
df ← data.frame(x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n))

# Scale for kmeans
df_s ← scale(df)

# Fit with K=4 and use cluster membership as a feature
km ← kmeans(df_s, centers = 4, nstart = 20)
df$cluster ← factor(km$cluster)

# Distance to each centroid also becomes a feature
dist_to_centroids ← as.matrix(dist(rbind(df_s, km$centers)))
dist_features ← dist_to_centroids[1:n, (n + 1):(n + 4)]
colnames(dist_features) ← paste0("dist_c", 1:4)

head(cbind(df, dist_features))
km$tot.withinss
```

13.10.6 Output & Results

The augmented data frame contains cluster membership plus four distance-to-centroid features. The model's WCSS quantifies clustering tightness; lower values indicate more compact clusters.

13.10.7 Interpretation

A reporting sentence: "k-means with $K = 4$ on the standardised features reduced WCSS from 900 (one cluster, equivalent to total variance) to 620 (four clusters); using cluster membership and distance-to-centroid as additional features in the downstream XGBoost

classifier improved cross-validated RMSE by 3 %." Always state K, the scaling, and the number of random starts.

13.10.8 Practical Tips

- Scale features before k-means; unscaled features with different units dominate the Euclidean distance and bias the partition.
- Use `nstart = 20` or higher to escape local minima; k-means is non-convex and a single start often produces sub-optimal solutions.
- k-means-style feature engineering works best when clusters actually exist in the data; on truly random data it adds noise features and can hurt downstream performance.
- Distance-to-centroid features are often more informative than the raw cluster label because they capture the continuous geometry that the discrete label discards.
- For high-dimensional data, reduce dimensionality with PCA before k-means; Euclidean distances concentrate and clusters lose meaning in high dimensions.
- For categorical or mixed data, k-means does not apply directly; use k-modes or PAM with Gower distance instead.

13.10.9 R Packages Used

Base R `kmeans()` for the canonical implementation; `factoextra::fviz_cluster()` for ggplot-style cluster visualisation; `klaR::kmodes()` for categorical alternatives; `recipes::step_kmeans_features()` (via `embed`) for tidymodels-flavoured pipeline integration; `cluster::clara()` for k-medoids alternatives that scale to large datasets.

13.10.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.10.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.11 Convolutional Neural Networks: Introduction

13.11.1 Introduction

Convolutional Neural Networks (CNNs; LeCun, 1989) replace fully-connected layers with local, weight-sharing convolutional filters. The shared weights encode translation invariance: a feature detector (edge, texture) learned once applies at every spatial location. CNNs dominate image classification, segmentation, and many signal-processing tasks.

13.11.2 Prerequisites

Feedforward networks; convolutions and cross-correlations; image tensors.

13.11.3 Theory

Convolutional layer: a set of learned filters K_c; each filter slides across the input, producing a feature map. Output channel count equals number of filters.

Pooling: max or average over a spatial window; reduces dimensionality and increases receptive field.

Typical architecture (classic): conv → ReLU → pool, repeated; then flatten to fully-connected for classification. Modern architectures (ResNet, EfficientNet) add residual connections and batch normalisation.

13.11.4 Assumptions

Inputs have spatial structure (images, spectrograms, time series); sufficient training data or a pretrained backbone.

13.11.5 R Implementation

```
library(torch)

torch_manual_seed(2026)

# Simulate a small batch of "images"
x ← torch_randn(4, 1, 28, 28)          # 4 images, 1 channel, 28x28
y ← torch_randint(0, 10, size = list(4), dtype = torch_long())

# Simple CNN: 2 conv layers + FC head
cnn ← nn_module(
  initialize = function() {
    self$conv1 ← nn_conv2d(1, 16, kernel_size = 3, padding = 1)
    self$conv2 ← nn_conv2d(16, 32, kernel_size = 3, padding = 1)
    self$pool  ← nn_max_pool2d(2)
    self$fc1   ← nn_linear(32 * 7 * 7, 64)
    self$fc2   ← nn_linear(64, 10)
  },
  forward = function(x) {
    x ← self$pool(nnf_relu(self$conv1(x)))
    x ← self$pool(nnf_relu(self$conv2(x)))
    x ← x$view(c(x$size(1), -1))
    nnf_relu(self$fc1(x)) %>% self$fc2()
  }
)
model ← cnn()
logits ← model(x)
logits$shape                # [4, 10]
```

13.11.6 Output & Results

Logits of shape [batch, classes] ready for a cross-entropy loss during training.

13.11.7 Interpretation

“A two-layer CNN produced 10-class logits for a batch of 28×28 inputs. In a real MNIST pipeline this architecture would reach ~99 % accuracy after training.”

13.11.8 Practical Tips

- Pre-trained CNN backbones (ResNet50, EfficientNet) with fine-tuning outperform training from scratch for most tasks with < 1M images.
- Data augmentation (flip, rotate, crop) is essential for image classification; use `torchvision::transform_*`.
- Batch normalisation accelerates training and stabilises gradients; include after every conv layer.
- For 1-D signals (ECG, audio), use 1-D convolutions (`nn_conv1d`).
- Vision transformers (ViT) now rival or exceed CNNs on large datasets; CNNs remain preferred for small-to-moderate data.

13.11.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.11.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.12 Cross-Validation

13.12.1 Introduction

Cross-validation is the primary tool for estimating how well a predictive model will generalise to unseen data. The in-sample error – the error of the fitted model on the data used to train it – is almost always too optimistic. Cross-validation repeatedly splits the data into training and held-out portions, fits the model on the training portion, evaluates on the held-out portion, and averages the held-out performance as an estimate of true generalisation error.

13.12.2 Prerequisites

The reader should understand the concepts of overfitting, bias and variance, and the training vs. test-set distinction. Familiarity with `tidymodels` is helpful but not assumed.

13.12.3 Theory

A model's **generalisation error** is its expected loss on a new observation drawn from the same population as the training data. The **training error** is its loss on the training data itself. The difference between them – the **optimism** of the training error – grows with model flexibility and shrinks with sample size.

13.12.3.1 k-fold cross-validation

The data are partitioned into k approximately equal folds. For each fold $j = 1, \ldots, k$, the model is fitted on the other $k - 1$ folds and evaluated on the held-out j-th fold. The k held-out losses are averaged. A common choice is $k = 5$ or $k = 10$, which balances bias

(too few folds means each training set is noticeably smaller than the full data) against variance (too many folds means the estimates are correlated).

13.12.3.2 Leave-one-out cross-validation (LOOCV)

The extreme case $k = n$: each fold has exactly one observation. LOOCV has low bias (the training sets are nearly the full data) but high variance and high computational cost. It is the right choice when n is small and the model is fast to fit.

13.12.3.3 Stratified cross-validation

For classification with class imbalance, folds are constructed to preserve class proportions. This reduces variability in the estimated performance and avoids folds with zero minority-class examples.

13.12.3.4 Repeated k-fold

k-fold CV is repeated with different random partitions, and the results averaged. This reduces Monte Carlo variability in the estimate without the computational cost of LOOCV.

13.12.3.5 Nested cross-validation

When hyperparameters are tuned on CV, the tuned performance estimate is optimistically biased because the hyperparameters have been chosen to maximise the same metric. Nested CV uses an outer loop for honest generalisation estimation and an inner loop for tuning: the outer folds never see the data used to choose the hyperparameters.

13.12.4 Assumptions

CV is a general technique, but it assumes:

1. The splits are representative of the data-generating process – the training and test folds come from the same distribution.
2. The observations are exchangeable (or the CV scheme respects the dependence structure – see below).
3. The performance metric is well-estimated by an average over folds.

For time-series or clustered data, standard CV leaks information between folds and overstates performance. Use time-series CV (expanding window) or group-blocked CV (by subject, site, etc.) in those cases.

13.12.5 R Implementation

```r
library(tidymodels)

data(penguins, package = "palmerpenguins")
penguins ← tidyr::drop_na(penguins)

set.seed(2026)
split ← initial_split(penguins, prop = 0.75, strata = species)
train ← training(split)
test  ← testing(split)

folds ← vfold_cv(train, v = 10, strata = species)

rec ← recipe(species ~ bill_length_mm + bill_depth_mm +
               flipper_length_mm + body_mass_g, data = train) ▷
  step_normalize(all_numeric_predictors())

spec ← multinom_reg() ▷ set_engine("nnet")

wf ← workflow() ▷ add_recipe(rec) ▷ add_model(spec)

cv_results ← fit_resamples(wf, resamples = folds,
                           metrics = metric_set(accuracy, roc_auc))

collect_metrics(cv_results)
```

This performs stratified 10-fold CV for a multinomial logistic regression on the `penguins` dataset. `fit_resamples()` handles the CV loop; `collect_metrics()` summarises the fold-level performance.

13.12.6 Output & Results

A typical output:

.metric	.estimator	mean	n	std_err
accuracy	multiclass	0.981	10	0.008
roc_auc	hand_till	0.999	10	0.000

The mean across folds is the point estimate; the standard error is the Monte Carlo uncertainty across folds.

13.12.7 Interpretation

The 10-fold cross-validated accuracy is 0.981 (SE 0.008), and the ROC AUC is 0.999 (SE < 0.001). These are estimates of out-of-sample performance on the penguin species classification task. The held-out test set is still unused; it is reserved for the final, honest evaluation after all modelling decisions are locked.

13.12.8 Practical Tips

- Preprocessing (scaling, imputation, encoding) must go *inside* the resampling loop to prevent leakage. `recipes` with `workflow()` does this correctly by default.

- Report both the mean and the standard error across folds so readers know how confident the estimate is.
- For hyperparameter tuning, use nested CV (or a train/validation/test split) – tuning on the same CV you report is biased.
- For time series, use `rolling_origin()` or `sliding_window()` in `rsample`; never use `vfold_cv()` on serial data.
- Small datasets have high-variance CV estimates – consider reporting a 95% CI across folds or using repeated CV.

13.12.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.12.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.13 Decision Trees (CART)

13.13.1 Introduction

Decision trees partition the feature space by recursive binary splits, fitting a constant prediction in each terminal region (leaf). The Classification and Regression Trees (CART) algorithm by Breiman, Friedman, Olshen and Stone (1984) is the dominant implementation. Trees are interpretable (a single fitted tree can be read as a sequence of if-else rules), handle mixed numeric and categorical data without preprocessing, require no feature scaling, and serve as the building blocks for random forests and gradient boosting. Their main weakness is high variance — small data perturbations produce very different trees — which motivates ensemble methods that aggregate many trees.

13.13.2 Prerequisites

A working understanding of classification and regression problems, impurity measures (Gini, entropy, variance), and the bias-variance trade-off in model complexity.

13.13.3 Theory

At each node, CART searches over all features and candidate split points, choosing the split that maximally reduces impurity:

- **Regression**: minimise within-node variance.
- **Classification**: minimise Gini $1 - \sum_k p_k^2$ or entropy $-\sum_k p_k \log p_k$.

Trees grow recursively until leaves are pure or a minimum-size constraint is hit, then are pruned back using cost-complexity pruning. The cost-complexity criterion is

$$R_\alpha(T) = R(T) + \alpha |T|,$$

where $R(T)$ is the training error and $|T|$ the number of terminal leaves. The pruning parameter α is chosen by cross-validation; the standard rule selects the smallest tree within one CV-standard-error of the minimum.

13.13.4 Assumptions

Splits are axis-aligned (a single tree captures interactions only by combining splits at different depths, which limits its ability to model rotated or oblique decision boundaries); observations are independent.

13.13.5 R Implementation

```
library(rpart); library(rpart.plot)

set.seed(2026)
n ← 400
df ← data.frame(
  x1 = runif(n), x2 = runif(n),
  y  = factor(ifelse(runif(n) < plogis(-2 + 3 * runif(n) + 2 * runif(n)),
                     "pos", "neg"))
)

tree ← rpart(y ~ x1 + x2, data = df,
             method = "class",
             control = rpart.control(cp = 0.001))

printcp(tree)                  # CV error vs complexity
tree_pruned ← prune(tree,
                    cp = tree$cptable[which.min(tree$cptable[, "xerror"]), "CP"])
rpart.plot(tree_pruned, box.palette = c("#F4A261", "#2A9D8F"))
```

13.13.6 Output & Results

`rpart()` returns a fitted tree object; `printcp()` reports the cross-validation error at each candidate complexity parameter; `prune()` produces the cost-complexity-pruned tree at the chosen α. `rpart.plot()` visualises the tree with split conditions and leaf class proportions.

13.13.7 Interpretation

A reporting sentence: “CART with cost-complexity pruning at the CV-optimal α retained 5 leaves; the dominant split was on x_1 at 0.52, partitioning 78 % of positive cases into one branch and 80 % of negatives into the other; sub-splits on x_2 refined predictions within each branch.” Always state the pruning method and the resulting tree size.

13.13.8 Practical Tips

- Always prune using the `xerror` column of the CP table; unpruned trees over-fit severely.
- A single tree has high variance; for predictive accuracy, prefer random forests, gradient boosting, or extra trees.
- Trees handle missing data via surrogate splits (rpart’s default); this is one of the few methods that does so without explicit imputation.
- Visualise with `rpart.plot()` for `ggplot`-friendly output; `partykit::as.party()` provides a richer object with print and plot methods.
- For regression, use `method = "anova"`; for classification, `method = "class"`; for survival, `method = "exp"` and the `rpart` package has built-in support.
- For interpretability with better accuracy, use a small ensemble (e.g., 50 trees in a random forest) and aggregate variable-importance measures from across trees.

13.13.9 R Packages Used

`rpart` for CART implementation with cost-complexity pruning; `rpart.plot` for tree visualisation; `partykit` for an alternative tree framework with richer output classes; `tree` for an older alternative implementation; `caret` and `tidymodels::decision_tree()` for cross-validated pruning within ML pipelines.

13.13.10 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.13.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.14 Dropout and Early Stopping

13.14.1 Introduction

Neural networks with many parameters easily overfit. Dropout (Srivastava et al., 2014) randomly zeros activations during training, acting as implicit ensembling. Early stopping halts training when validation loss stops improving, preventing the model from memorising training noise. Both are simple, cheap, and among the most effective regularisation techniques for deep nets.

13.14.2 Prerequisites

Feedforward networks; validation-based evaluation.

13.14.3 Theory

Dropout: during training, each neuron's activation is set to zero with probability p (typically 0.2-0.5); during inference, activations are scaled by $1 - p$ or equivalent. Acts like training an ensemble of 2^n sub-networks with shared weights.

Early stopping: train on training set, monitor a validation loss; halt when validation loss stops improving for a "patience" window. Implicitly regularises – the final model has parameters closer to the initialisation than full convergence would reach.

13.14.4 Assumptions

A held-out validation set exists; model is over-parameterised enough to overfit without regularisation.

13.14.5 R Implementation

```
library(torch)

torch_manual_seed(2026)

# Model with dropout after each hidden layer
net <- nn_module(
  initialize = function() {
    self$fc1  <- nn_linear(10, 64)
    self$drop <- nn_dropout(p = 0.3)
    self$fc2  <- nn_linear(64, 1)
  },
  forward = function(x) {
    x %>% self$fc1() %>% nnf_relu() %>% self$drop() %>% self$fc2()
  }
)
model <- net()
opt   <- optim_adam(model$parameters, lr = 1e-2)

n <- 500
x_tr <- torch_randn(n, 10); y_tr <- torch_randn(n, 1)
x_va <- torch_randn(200, 10); y_va <- torch_randn(200, 1)

best_val <- Inf; patience <- 5; counter <- 0
for (epoch in 1:60) {
  model$train()
  opt$zero_grad()
  loss <- nnf_mse_loss(model(x_tr), y_tr)
  loss$backward(); opt$step()

  model$eval()
  val <- nnf_mse_loss(model(x_va), y_va)$item()
  if (val < best_val - 1e-4) { best_val <- val; counter <- 0 }
  else { counter <- counter + 1; if (counter >= patience) break }
}
cat("Stopped at epoch", epoch, "  best_val=", round(best_val, 3), "\n")
```

13.14.6 Output & Results

Training halts when validation loss has stopped improving for 5 epochs; dropout noise in the forward pass further regularises the model.

13.14.7 Interpretation

“Combining 30 % dropout with early stopping (patience 5) halted training at epoch 23 with best validation MSE 0.98, preventing the over-fitting observed beyond epoch 30 without regularisation.”

13.14.8 Practical Tips

- Dropout probability 0.2-0.5 for hidden layers; avoid dropout on output layers.
- Dropout is automatically disabled in `eval()` mode; don’t forget to switch modes.
- Use weight decay (L2 on weights) jointly with dropout for stronger regularisation.
- Early stopping is the cheapest regulariser; always implement it for deep nets.

- For batch norm + dropout, place dropout after batch norm; order matters for training stability.

13.14.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.14.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.15 Extremely Randomised Trees

13.15.1 Introduction

Extremely Randomised Trees — usually shortened to "Extra Trees" — are a variant of random forests introduced by Geurts, Ernst, and Wehenkel in 2006. The defining innovation: instead of optimising the split threshold at each node, Extra Trees draws the threshold *at random* from the feature's range. The additional source of randomness increases bias slightly but reduces variance further, often producing models with accuracy competitive to random forests at a fraction of the training time.

13.15.2 Prerequisites

A working understanding of random forests, decision-tree splitting criteria (Gini, variance), and the bias-variance trade-off in ensemble methods.

13.15.3 Theory

At each node, Extra Trees:

1. Choose m random features (as in random forest).
2. For each chosen feature, draw a *random threshold* uniformly from its observed range — not the optimal split.

3. Split on whichever (feature, random-threshold) pair gives the lowest impurity among these candidates.

Trees are typically fit to the entire training data without bootstrap sampling (`replace = FALSE`); variance reduction comes from the random thresholds instead of the bootstrap. The combination of feature subsampling and threshold randomisation produces highly decorrelated trees, and averaging over many of them produces a low-variance ensemble.

13.15.4 Assumptions

Feature importance is approximately diffuse — random thresholds work poorly when a few dominant informative features need precise split points; in those settings, optimising thresholds (random forest) outperforms.

13.15.5 R Implementation

```
library(ranger)

set.seed(2026)
n ← 500
df ← data.frame(
  x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n), x4 = rnorm(n),
  y  = sin(2 * rnorm(n)) + 0.3 * rnorm(n)
)

# ranger implements extra-trees via `splitrule = "extratrees"`
rf ← ranger(y ~ ., data = df,
            num.trees = 500, mtry = 2,
            splitrule = "variance")

et ← ranger(y ~ ., data = df,
            num.trees = 500, mtry = 2,
            splitrule = "extratrees",
            num.random.splits = 5)

c(rf_oob_rmse = sqrt(rf$prediction.error),
  et_oob_rmse = sqrt(et$prediction.error))
```

13.15.6 Output & Results

OOB-based RMSE for both random forest and Extra Trees on the same data. Extra Trees typically produces slightly lower RMSE on noisy datasets where variance reduction dominates; random forest typically wins on structured signals where threshold optimisation captures sharper boundaries.

13.15.7 Interpretation

A reporting sentence: “Extra Trees with `num.random.splits = 5` produced marginally lower OOB RMSE (0.38) than random forest (0.41) on this noisy regression task; the variance reduction from random thresholds outweighed the small bias cost. Training was approximately 3 × faster than random forest.” Always state the number of random splits and compare against random forest baseline.

13.15.8 Practical Tips

- Extra Trees can skip bootstrapping (`replace = FALSE`) for tighter variance; random forest keeps the bootstrap as its primary variance-reduction mechanism.
- With many informative features, tune `num.random.splits` upward (10–20) so the random-threshold candidates include the optimum more often; this approaches random forest behaviour.
- Training is typically 2–5× faster than random forest for the same tree count, because no threshold-optimisation search is performed at each split.
- Use random forest when a few features dominate the signal; use Extra Trees when signal is diffuse across many features.
- For variable importance, use permutation importance (`importance = "permutation"`); impurity-based importance is biased in Extra Trees just as in random forests.
- For very large datasets, Extra Trees' speed advantage compounds; consider it as the default ensemble when training time matters.

13.15.9 R Packages Used

`ranger` with `splitrule = "extratrees"` for the canonical implementation; `extraTrees` package for the Python-compatible reference implementation; `parsnip::rand_forest()` (with engine selection) for tidymodels integration; `caret` for cross-validated tuning of `num.random.splits`.

13.15.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.15.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.16 Feature Engineering

13.16.1 Introduction

Feature engineering – creating informative inputs – is often the largest driver of predictive performance. Even deep learning benefits from domain-informed features. Standard techniques: log/sqrt transforms, interactions, polynomial bases, categorical encodings, date features, and lag features for time series.

13.16.2 Prerequisites

Regression / classification basics; understanding the target's scale and distribution.

13.16.3 Theory

Key transformations: - **Log / sqrt** stabilise right-skewed variables. - **Box-Cox / Yeo-Johnson** learn the optimal power transform. - **Polynomial and spline bases** expose non-linear relationships to linear models. - **Interactions** ($x_1 x_2$) surface effects whose direction depends on another variable. - **Target encoding** for high-cardinality categoricals: replace category with smoothed target mean. - **Lag / rolling features** for time series.

13.16.4 Assumptions

Transformations preserve monotonicity where needed; target encoding uses out-of-fold means to avoid leakage.

13.16.5 R Implementation

```
library(recipes); library(embed)

set.seed(2026)
n <- 300
df <- tibble::tibble(
  x1 = exp(rnorm(n)),                    # right-skewed
  x2 = runif(n, 0, 10),
  cat = sample(letters[1:20], n, replace = TRUE),
  date = as.Date("2024-01-01") + sample(0:364, n, replace = TRUE),
  y = runif(n)
)

rec <- recipe(y ~ ., data = df) %>%
  step_log(x1) %>%
  step_interact(~ x1:x2) %>%
  step_ns(x2, deg_free = 4) %>%
  step_date(date, features = c("dow", "month")) %>%
  step_holiday(date) %>%
  step_rm(date) %>%
  step_lencode_mixed(cat, outcome = vars(y))   # mixed-effects target encoding

baked <- prep(rec) %>% bake(new_data = NULL)
colnames(baked)[1:10]
```

13.16.6 Output & Results

A transformed data frame with log-x1, interaction term, natural spline basis for x2, day-of-week / month features, and smoothly target-encoded category.

13.16.7 Interpretation

“Feature engineering added 12 derived columns: log-transformed skew, spline bases, date features, and target-encoded category. The pipeline is reproducible and applied identically to train and test.”

13.16.8 Practical Tips

- Do feature engineering inside a `recipe` (or equivalent) to avoid leakage.
- Log-transform right-skewed variables before linear models; trees handle skew fine.
- High-cardinality categoricals benefit hugely from target encoding (with proper smoothing / CV handling).
- Date features (day-of-week, hour, month, holiday) are often critical for temporal prediction.
- Don’t over-engineer – more features means more noise; cross-validate additions.

13.16.9 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.16.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.17 Feature Selection

13.17.1 Introduction

Feature selection retains the predictors most relevant to the target while discarding the rest. The benefits are simpler models, reduced over-fitting, faster inference, improved interpretability, and reduced annotation cost in some applications. Three families of

methods coexist: filter (univariate tests, cheap and fast), wrapper (fit-and-evaluate, slow but thorough), and embedded (selection happens during fitting, balancing speed and quality). Modern ML pipelines almost always use embedded methods (lasso, tree importance, Boruta) because they integrate naturally with cross-validation.

13.17.2 Prerequisites

A working understanding of regression and classification basics, regularisation (L1/lasso), and the bias-variance trade-off.

13.17.3 Theory

Filter methods rank features by univariate statistics — Pearson or Spearman correlation, mutual information, chi-squared, ANOVA F-statistic — and select the top k. Fast but ignore feature interactions and joint information.

Wrapper methods treat the model as a black box, fitting and evaluating across feature subsets. Forward selection adds features one at a time; backward elimination starts with all features and removes them; recursive feature elimination (RFE) cycles through both with cross-validation. Slow but capable of finding complementary feature combinations.

Embedded methods integrate selection with fitting. Lasso drives some coefficients to exactly zero (selection); tree-based methods (random forest, XGBoost) provide importance scores; Boruta compares each feature against random permuted shadows in many resamples to test for genuinely predictive features. Stability selection runs lasso on many subsamples and ranks features by selection frequency.

13.17.4 Assumptions

Selected features generalise from training to deployment data; selection is done within CV folds to prevent leakage of test-fold information into the chosen feature set.

13.17.5 R Implementation

```r
library(Boruta); library(glmnet)

set.seed(2026)
n ← 300; p ← 20
X ← matrix(rnorm(n * p), n, p)
colnames(X) ← paste0("x", 1:p)
y ← X %*% c(rep(2, 5), rep(0, 15)) + rnorm(n)

# Embedded: Lasso selects non-zero features
lasso ← cv.glmnet(X, y, alpha = 1)
nonzero_lasso ← rownames(coef(lasso, s = "lambda.min"))[
  which(as.vector(coef(lasso, s = "lambda.min")) ≠ 0)
][-1]
cat("Lasso selected:", length(nonzero_lasso), "\n")

# Wrapper-ish: Boruta compares features to shadow permutations
bor ← Boruta(X, y, maxRuns = 50, doTrace = 0)
confirmed ← getSelectedAttributes(bor, withTentative = FALSE)
cat("Boruta confirmed:", length(confirmed), "\n")
```

13.17.6 Output & Results

Lasso retains the features whose coefficients survive the L1 penalty at the cross-validation-optimal λ. Boruta confirms or rejects features based on whether their importance exceeds that of randomly permuted shadow features across many resampling runs. On the sparse-signal dataset (5 true features, 15 noise), both methods typically recover 4–5 true features with near-zero false positives.

13.17.7 Interpretation

A reporting sentence: "Lasso (cross-validated $\lambda = 0.08$) and Boruta (50 runs, BH-adjusted importance against shadow features) independently identified the same 5 predictive features matching the data-generating model, with no false positives in either; correlation-based filter selection (top 5 by absolute correlation with y) missed one feature because its marginal correlation was masked by noise from collinear predictors." Always describe the selection method and any cross-validation embedding.

13.17.8 Practical Tips

- Perform feature selection *inside* cross-validation folds; selection on the full dataset before CV leaks information from test folds into the chosen feature set.
- Lasso is fast and principled for linear or near-linear relationships; random forest importance handles non-linear effects better.
- Boruta's multiple-run shadow-feature comparison is statistically cleaner than single-pass importance and provides confidence statements about selection.
- For $p > n$ regimes, lasso (or elastic net) is nearly mandatory; random forest importance alone becomes unreliable in very high-dimensional settings.
- Stability selection (`stabs::stabsel`) fits many lassos on data subsamples and ranks features by their selection frequency; this is a robust alternative when a single CV run is unstable.

- Pre-screen with cheap filter methods (variance, correlation) when p is in the millions; this reduces the candidate set before applying the more expensive wrapper or embedded methods.

13.17.9 R Packages Used

`glmnet` for lasso and elastic-net embedded selection; `Boruta` for shadow-feature-based confirmation; `stabs` for stability selection; `caret::rfe()` for recursive feature elimination; `vip` for permutation-based variable importance from any model; `recipes::step_select_*` for tidymodels-flavoured selection within preprocessing pipelines.

13.17.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.17.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.18 Feedforward Networks in torch

13.18.1 Introduction

`torch` for R is the native R port of PyTorch: an automatic-differentiation library with GPU support, a comprehensive layer library, and modern optimisers (Adam, AdamW). It enables building arbitrary neural-network architectures from scratch and training them with the same APIs Python users know. The companion package `luz` wraps `torch` with a scikit-learn-style `fit` / `predict` interface, abstracting away the manual training loop for routine cases.

13.18.2 Prerequisites

A working understanding of neural networks (forward propagation, backpropagation), gradient descent, the chain rule, and basic mini-batch training concepts.

13.18.3 Theory

A `torch` model is an `nn_module` — a composition of parameterised layers with a forward-pass method. Training iterates over epochs and mini-batches:

1. **Forward pass**: compute predictions and loss on a mini-batch.
2. **Backward pass**: compute gradients via reverse-mode automatic differentiation.
3. **Optimiser step**: update parameters using the gradient (Adam adjusts step sizes per parameter).

Modern training loops add learning-rate scheduling (cosine annealing, warmup), gradient clipping, weight decay, and early stopping. `luz` packages these patterns into named callbacks.

13.18.4 Assumptions

Sufficient data for the chosen model capacity (overparameterised models on tiny datasets memorise rather than generalise); features standardised before input; mini-batch size appropriate for available memory and the bias-variance trade-off.

13.18.5 R Implementation

```
library(torch)

set.seed(2026)
torch_manual_seed(2026)

# Simulate a simple regression task
n ← 1000
x ← torch_randn(n, 3)
y ← (x[, 1] * 2 + x[, 2]^2 - x[, 3] * 0.5)$unsqueeze(2)

net ← nn_module(
  initialize = function() {
    self$fc1 ← nn_linear(3, 16)
    self$fc2 ← nn_linear(16, 16)
    self$fc3 ← nn_linear(16, 1)
  },
  forward = function(x) {
    x %>% self$fc1() %>% nnf_relu() %>%
          self$fc2() %>% nnf_relu() %>%
          self$fc3()
  }
)

model ← net()
opt   ← optim_adam(model$parameters, lr = 1e-2)

for (epoch in 1:50) {
  opt$zero_grad()
  pred ← model(x)
  loss ← nnf_mse_loss(pred, y)
  loss$backward()
  opt$step()
  if (epoch %% 10 == 0)
    cat(sprintf("epoch %d  loss %.4f\n", epoch, loss$item()))
}
```

13.18.6 Output & Results

Per-epoch MSE loss should decrease monotonically as the network learns the non-linear regression. The 3-layer MLP with ReLU activations rapidly fits the simulated quadratic-in-x_2 structure that linear regression cannot capture.

13.18.7 Interpretation

A reporting sentence: "A 3-layer feedforward network (16-16-1 hidden units, ReLU activations, Adam optimiser, learning rate 0.01) trained for 50 epochs reduced MSE from 3.1 to 0.04 on the simulated regression with quadratic interactions; predictions closely tracked the noisy targets, with residuals consistent with the irreducible simulation noise level." Always state the architecture, optimiser, learning rate, and training epochs.

13.18.8 Practical Tips

- Use `luz::fit()` for a scikit-learn-style interface that handles the training loop, dataloaders, and standard callbacks (early stopping, model checkpointing, learning-rate scheduling).
- Move tensors and models to GPU with `$to(device = "cuda")`; check `cuda_is_available()` first to handle CPU-only systems gracefully.
- Mini-batching via `dataloader()` is essential beyond toy problems; full-batch gradient descent is computationally inefficient and produces noisier optimisation trajectories than expected.
- Monitor training and validation loss separately; early-stop when validation loss plateaus or starts to rise to prevent over-fitting.
- `torchvision` and `torchaudio` add pre-trained models for transfer learning; these dramatically reduce training time and data requirements for common tasks (image classification, audio recognition).
- For tabular data, gradient-boosted trees (XGBoost, LightGBM) usually outperform feedforward networks; deep learning shines on raw modalities (images, text, audio) where structure is encoded in the data layout.

13.18.9 R Packages Used

`torch` for the canonical PyTorch-like API in R; `luz` for sklearn-style `fit` / `predict`; `torchvision`, `torchaudio`, `tabnet` for domain-specific models; `torcheval` for evaluation metrics; `coro` for Python-style generators (used internally by `dataloader`).

13.18.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.18.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.19 Gradient Boosting

13.19.1 Introduction

Gradient boosting, formalised by Friedman in 2001, builds an additive ensemble $F_M(x) = \sum_{m=1}^{M} \nu h_m(x)$ by fitting each weak learner h_m to the negative gradient of the loss at the current predictions. Decision trees are the canonical weak learner, the learning rate ν shrinks each tree's contribution to control regularisation, and the resulting framework dominates structured tabular-data benchmarks. Modern implementations (XGBoost, LightGBM, CatBoost) add second-order information, regularisation, missing-value handling, and other improvements over the original Friedman gradient boosting.

13.19.2 Prerequisites

A working understanding of decision trees, gradient descent, and the bias-variance trade-off in ensemble methods.

13.19.3 Theory

The generic gradient-boosting algorithm:

1. Initialise $F_0(x) = \arg\min_\gamma \sum_i L(y_i, \gamma)$ — the constant prediction minimising the loss.
2. For $m = 1, \dots, M$:
   - Compute pseudo-residuals $r_{im} = -\partial L(y_i, F(x_i))/\partial F(x_i)|_{F=F_{m-1}}$.
   - Fit a weak learner h_m to the pseudo-residuals.
   - Update $F_m(x) = F_{m-1}(x) + \nu h_m(x)$.

For squared-error loss the pseudo-residuals are ordinary residuals; for binary log-loss they are score residuals; for Huber loss they combine the two regimes. The learning rate ν trades fit quality for over-fit risk: smaller ν requires more trees but generalises better.

13.19.4 Assumptions

The chosen weak learner is appropriate for the loss (shallow trees with depth 3–6 for most losses); the learning rate is small enough for stable optimisation; sufficient sample size for the boosting interactions to be supported by the data.

13.19.5 R Implementation

```
library(gbm)

set.seed(2026)
n ← 500
df ← data.frame(
  x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n),
  y  = sin(rnorm(n)) + 0.5 * rnorm(n)^2
)

m ← gbm(y ~ ., data = df,
         distribution = "gaussian",
         n.trees = 2000,
         interaction.depth = 3,
         shrinkage = 0.01,
         cv.folds = 5,
         verbose = FALSE)

best_iter ← gbm.perf(m, method = "cv", plot.it = FALSE)
cat("Best iter:", best_iter, "\n")
summary(m, n.trees = best_iter, plotit = FALSE)
```

13.19.6 Output & Results

`gbm()` with `cv.folds` returns the cross-validated boosting iteration count along with the variable-importance summary. `gbm.perf()` finds the iteration that minimises CV error; predictions at that iteration count are typically used for deployment.

13.19.7 Interpretation

A reporting sentence: “Gradient boosting with `shrinkage = 0.01`, `interaction.depth = 3`, and 5-fold cross-validated early stopping converged at 1{,}240 trees (CV RMSE 0.34); variable-importance scores attributed 52 % of relative influence to x_1, 28 % to x_2, and 20 % to x_3, consistent with the simulated data structure.” Always state the shrinkage, depth, CV-optimal iteration count, and importance scores.

13.19.8 Practical Tips

- Smaller learning rate (0.01–0.05) with many trees almost always generalises better than larger rate with few trees; this is the canonical regularisation trade-off in boosting.
- Use shallow trees (depth 3–6) as the standard weak learners; deeper trees approach a single-tree model and lose the boosting benefit.
- `cv.folds = 5` in `gbm()` produces an early-stopping estimate without separate plumbing; `gbm.perf(method = "cv")` finds the optimal iteration count.

- For structured prediction problems, prefer XGBoost, LightGBM, or CatBoost over base `gbm`; they add regularisation, missing-value handling, GPU support, and substantially faster training.
- Monotone constraints (supported by XGBoost and LightGBM) embed domain knowledge into boosting; useful in regulated industries (credit, insurance, healthcare).
- Pair gradient boosting with feature-importance and partial-dependence plots for interpretation; raw boosted trees are otherwise opaque.

13.19.9 R Packages Used

`gbm` for the original Friedman gradient boosting; `xgboost`, `lightgbm`, `catboost` for modern alternatives with regularisation and speed; `parsnip::boost_tree()` for tidymodels integration; `mboost` for component-wise gradient boosting on additive models.

13.19.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.19.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.20 Isolation Forest

13.20.1 Introduction

Isolation Forest (Liu, Ting, and Zhou, 2008) is a tree-based unsupervised anomaly-detection method built on a single intuition: outliers are easier to isolate by random splits than densely packed normal points. The algorithm builds many random trees, partitions the data with random feature-and-split-value choices, and uses each observation's average path length to a terminal node as its anomaly score. Short paths mean easy isolation, which means the point is anomalous; long paths mean the point sits in dense regions that take many splits to isolate.

13.20.2 Prerequisites

A working understanding of decision trees, random partitioning, and basic anomaly-detection concepts.

13.20.3 Theory

Build T isolation trees: at each node, pick a random feature and a random split value uniformly in $[\min, \max]$ for that feature. Tree depth is capped at $\log_2 n$ for computational efficiency. For an observation x, the anomaly score is

$$s(x) = 2^{-\mathbb{E}[h(x)]/c(n)},$$

where $h(x)$ is the average path length across trees and $c(n)$ is a normalising constant equal to the average path length of an unsuccessful BST search on n points. Scores close to 1 indicate clear anomalies; scores around 0.5 indicate normal points; scores well below 0.5 are deep in dense regions.

13.20.4 Assumptions

Anomalies are few and distinct from the dense normal points; the subsampling size is sufficient to isolate them ($\psi = 256$ is the canonical default and works well in practice).

13.20.5 R Implementation

```
library(solitude)

set.seed(2026)
n ← 300
df ← data.frame(
  x1 = c(rnorm(n), runif(20, -5, 5)),
  x2 = c(rnorm(n), runif(20, -5, 5))
)

iforest ← isolationForest$new(num_trees = 200)
iforest$fit(df)

scores ← iforest$predict(df)$anomaly_score
flagged ← scores > quantile(scores, 0.9)

plot(df, col = ifelse(flagged, "red", "grey60"),
     pch = 16, asp = 1,
     main = "Isolation Forest: top-10% scored observations flagged",
     xlab = "x1", ylab = "x2")
```

13.20.6 Output & Results

`isolationForest$predict()` returns a tibble with anomaly scores per observation. Thresholding at a chosen quantile (e.g., the 90th percentile) flags the candidates; the scatter shows flagged points in red and normal points in grey.

13.20.7 Interpretation

A reporting sentence: “Isolation Forest with 200 trees and default subsampling scored 14 of 20 injected anomalies (sensitivity 0.70) at the 90th-percentile threshold, with 16 false positives among the 300 normal points (false-positive rate 5.3 %); the score distribution showed a clear bimodal structure separating anomalies from normal points.” Always state the number of trees, threshold, and per-class accuracy.

13.20.8 Practical Tips

- Tune the contamination threshold (top-k percentile) to match the expected anomaly rate; a 5 % threshold for typical fraud detection, 1 % for very rare events.
- `num_trees = 100` - `200` is usually sufficient; returns diminish beyond 200 and computational cost grows linearly.
- Isolation Forest handles high-dimensional data well because it does not rely on distance computations; this is one of its main advantages over distance-based methods (kNN-distance, LOF).
- Excellent default choice for anomaly detection; requires no feature scaling and has few hyperparameters to tune.
- For improved separation in non-axis-aligned anomaly structures, Extended Isolation Forest adds random oblique splits; available via the `isotree` package.
- Pair anomaly scores with domain expertise; flagged observations should be investigated, not automatically removed or actioned.

13.20.9 R Packages Used

`solitude` for the canonical R6-based isolation forest interface; `isotree` for Extended Isolation Forest with oblique splits and additional anomaly-detection variants; `dbscan::lof()` for Local Outlier Factor as a distance-based alternative; `mlr3` and `tidymodels` for cross-validated anomaly detection with labelled data.

13.20.10 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.20.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro

- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.21 Isotonic Regression Calibration

13.21.1 Introduction

Isotonic regression fits a monotone non-decreasing step function from classifier scores to observed probabilities. It is the non-parametric counterpart to Platt scaling: more flexible (no sigmoid assumption required) but requiring more validation data and more prone to over-fitting. It is the standard post-hoc calibration method for tree-ensemble classifiers (random forest, gradient boosting), whose mis-calibration patterns are often non-sigmoidal.

13.21.2 Prerequisites

A working understanding of calibration plots, the pool-adjacent-violators (PAV) algorithm, and Platt scaling as the parametric alternative.

13.21.3 Theory

Given sorted score-label pairs (f_i, y_i), isotonic regression finds the non-decreasing step function $\hat{g}$ minimising $\sum_i (y_i - \hat{g}(f_i))^2$. The pool-adjacent-violators algorithm solves this in $O(n)$ time: it walks through the sorted scores, pooling adjacent observations into single steps whenever monotonicity is violated, until the result is non-decreasing.

The fitted step function maps any future score to the observed probability rate within the matching score range from the validation set. The non-parametric flexibility allows fitting any monotone calibration curve, but at the cost of statistical efficiency relative to Platt's two-parameter sigmoid.

13.21.4 Assumptions

The mis-calibration is monotone (almost always the case for well-trained classifiers); enough validation data to estimate the step function without over-fitting (rule of thumb: at least 500 examples).

13.21.5 R Implementation

```
set.seed(2026)
n ← 1000
y ← rbinom(n, 1, 0.3)
f ← 2 * y + rnorm(n, 0, 1.2)

val_idx ← 1:500
te_idx  ← 501:1000

# Isotonic regression on validation; apply to test
iso ← isoreg(f[val_idx], y[val_idx])
iso_fn ← stepfun(sort(f[val_idx]), c(0, iso$yf))
iso_p  ← iso_fn(f[te_idx])
iso_p  ← pmin(pmax(iso_p, 0), 1)

naive_p ← plogis(f[te_idx])
platt   ← glm(y[val_idx] ~ f[val_idx], family = "binomial")
platt_p ← plogis(coef(platt)[1] + coef(platt)[2] * f[te_idx])

c(naive = mean((naive_p - y[te_idx])^2),
  platt = mean((platt_p - y[te_idx])^2),
  isotonic = mean((iso_p  - y[te_idx])^2))
```

13.21.6 Output & Results

Brier scores for the three methods: naive sigmoid (no calibration), Platt scaling (parametric sigmoid), and isotonic regression (non-parametric step function). Isotonic typically wins when the underlying calibration curve is non-sigmoidal.

13.21.7 Interpretation

A reporting sentence: “Isotonic regression on the held-out test set produced the lowest Brier score (0.168) compared with Platt scaling (0.181) and the naive sigmoid (0.213); isotonic captured a non-sigmoidal calibration curve that the parametric sigmoid could not. ROC AUC was unchanged across all three methods because the rescaling preserves rank.” Always compare both calibration methods and report Brier score plus a reliability diagram.

13.21.8 Practical Tips

- Isotonic regression needs more validation data than Platt scaling (rule of thumb: at least 500 examples; for very imbalanced data, more for the minority class).
- Prone to over-fit when the validation set is small; the step function can track validation-specific noise, especially in the tails. Cross-validate the calibration step itself for small validation sets.
- For tree-ensemble classifiers (random forest, GBM, XGBoost), isotonic regression is usually preferred over Platt; their mis-calibration is rarely sigmoidal.
- For SVMs, Platt scaling is usually sufficient; the SVM decision-value distribution is closer to logistic.
- Step-function output produces piecewise-constant probabilities, which can look strange in sensitive applications; spline-smoothed isotonic exists but is rarely implemented in production.

- Always pair calibration with a reliability diagram (`probably::cal_plot`, `gbm::calibrate.plot`) showing pre- and post-calibration curves; verify visually that the calibration improved.

13.21.9 R Packages Used

Base R `isoreg()` for the canonical PAV-based isotonic regression; `caret::calibrate()` and `probably::cal_estimate_isotonic()` for integrated tidymodels-style calibration; `betacal` for the parametric Beta calibration alternative; `mlr3calibration` for calibration extensions in mlr3.

13.21.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.21.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.22 k-Nearest Neighbours Classification

13.22.1 Introduction

k-nearest-neighbours classification assigns each test observation the majority class among its k nearest training points in feature space. It is the simplest instance-based learning method: there is no training step beyond storing the data, and prediction is a search-and-vote operation. kNN is competitive on low-dimensional, well-separated data and serves as a useful baseline against which more complex methods are compared. Performance depends critically on the distance metric and on k, both of which require attention.

13.22.2 Prerequisites

A working understanding of distance metrics (Euclidean, Manhattan, Mahalanobis), train-test splits, and the bias-variance trade-off in supervised learning.

13.22.3 Theory

For a query point x^* and training set $\{(x_i, y_i)\}$, the prediction is the mode of the labels of the k nearest neighbours: $\hat{y}(x^*) = \text{mode}\{y_i : x_i \in N_k(x^*)\}$. Distance-weighted variants weight votes inversely by distance, giving closer neighbours more influence and often improving boundary smoothness.

The bias-variance trade-off in k is direct: small k produces high-variance, low-bias decisions (each prediction depends on a few neighbours and adapts to local noise); large k produces high-bias, low-variance decisions (each prediction averages over a larger neighbourhood and misses local structure). Cross-validation chooses the optimal k.

The choice of distance metric matters: Euclidean is the default, Manhattan is more robust to outliers, Mahalanobis accounts for feature covariance, and Gower distance handles mixed numeric-categorical inputs.

13.22.4 Assumptions

Features are appropriately scaled (distances are scale-sensitive); labels are sufficiently clean for local majority voting to be informative; the chosen distance metric reflects meaningful similarity.

13.22.5 R Implementation

```
library(class)

set.seed(2026)
n ← 400
x1 ← rnorm(n); x2 ← rnorm(n)
y ← factor(ifelse(x1^2 + x2^2 < 1, "in", "out"))
df ← data.frame(x1 = scale(x1)[, 1], x2 = scale(x2)[, 1], y = y)

idx ← sample(n, n * 0.7)
train ← df[idx, ]; test ← df[-idx, ]

# kNN at different k
accs ← sapply(c(1, 5, 15, 51), function(k) {
  pred ← knn(train[, 1:2], test[, 1:2], train$y, k = k)
  mean(pred == test$y)
})
setNames(accs, paste0("k=", c(1, 5, 15, 51)))
```

13.22.6 Output & Results

Test accuracy across a grid of k values reveals the U-shaped bias-variance curve: very small k overfits training noise, very large k oversmooths the decision boundary, and a moderate k minimises test error.

13.22.7 Interpretation

A reporting sentence: "k-NN classification achieved best test accuracy of 0.91 at $k = 15$ on the circular-decision-boundary task; $k = 1$ over-fit to noise with accuracy 0.82, while $k = 51$ over-smoothed and dropped to 0.80. Cross-validation across k confirmed $k = 15$ as the optimum." Always state the chosen k, the distance metric, and the test-set accuracy.

13.22.8 Practical Tips

- Always standardise features before kNN; unscaled features with different units dominate the distance computation and bias predictions.
- Tune k by cross-validation; the U-shaped accuracy-vs-k curve has a clear optimum on most datasets.
- For high-dimensional data ($d > 20$), kNN suffers the curse of dimensionality: distances concentrate and discriminating power collapses. Reduce dimensionality with PCA or use a parametric / tree-based model.
- `kknn::train.kknn()` implements kernel-weighted kNN (closer neighbours weighted more heavily); often slightly more accurate than uniform-weight kNN.
- kNN is a lazy learner: prediction cost is $O(nd)$ per query for naive search; use `FNN` with kd-tree or ball-tree indexing for $O(\log n)$ queries on large training sets.
- For mixed numeric-categorical inputs, use Gower distance via `cluster::daisy()` instead of Euclidean; the standard distance breaks down on categorical features.

13.22.9 R Packages Used

`class::knn()` for the canonical implementation; `FNN::knn()` for fast kd-tree-based search; `kknn::train.kknn()` for weighted kNN with cross-validated k and kernel selection; `caret` and `tidymodels` for cross-validated kNN within ML pipelines; `RANN` for approximate nearest-neighbour search at scale.

13.22.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.22.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines

- FDR, knockoffs, replication crisis

13.23 k-Nearest Neighbours Regression

13.23.1 Introduction

k-Nearest Neighbours regression predicts a continuous outcome at a query point as the mean (or weighted mean) of the k nearest training targets in feature space. It is the simplest non-parametric regression method, making no assumption about the global functional form and only requiring a distance metric. The trade-off is straightforward: small k produces high variance (each prediction depends on a few neighbours), while large k produces high bias (predictions average over a wide region and miss local structure).

13.23.2 Prerequisites

A working understanding of distance metrics, the bias-variance trade-off, and train-test evaluation in supervised learning.

13.23.3 Theory

The kNN regression prediction at query point x^* is

$$\hat{y}(x^*) = \frac{1}{k} \sum_{i \in N_k(x^*)} y_i,$$

where $N_k(x^*)$ is the set of indices of the k training points nearest to x^*. Distance-weighted variants use weights $w_i \propto 1/\|x^* - x_i\|$ to give closer neighbours more influence than far ones, producing smoother predictions and reducing the discontinuities that plain kNN exhibits at the boundaries between Voronoi cells.

The bias-variance trade-off in k is direct: $k = 1$ memorises training noise (high variance, low bias); $k = n$ predicts the global mean (low variance, high bias). Cross-validation chooses the optimal k for the data.

13.23.4 Assumptions

Features are appropriately scaled (unscaled features with different units dominate the distance computation); the regression surface is locally smooth (kNN cannot extrapolate or model abrupt changes well).

13.23.5 R Implementation

```r
library(FNN)

set.seed(2026)
n <- 400
x <- runif(n, 0, 2 * pi)
y <- sin(x) + rnorm(n, 0, 0.3)
df <- data.frame(x = scale(x)[, 1], y = y)

idx <- sample(n, n * 0.7)
train <- df[idx, ]; test <- df[-idx, ]

rmses <- sapply(c(1, 5, 15, 51), function(k) {
  pred <- knn.reg(train = train[, "x", drop = FALSE],
                  test  = test[,  "x", drop = FALSE],
                  y = train$y, k = k)$pred
  sqrt(mean((test$y - pred)^2))
})
setNames(rmses, paste0("k=", c(1, 5, 15, 51)))
```

13.23.6 Output & Results

Test RMSE across a grid of k values reveals the U-shaped bias-variance curve: very small k overfits noise, very large k oversmooths the sinusoid, and a moderate k (around 15 in this example) minimises test error.

13.23.7 Interpretation

A reporting sentence: "k-NN regression with $k = 15$ achieved test RMSE of 0.31 on the held-out 30 % of the simulated sinusoid, matching the irreducible noise level ($\sigma = 0.3$); $k = 1$ over-fitted to RMSE 0.42, $k = 51$ under-fitted to RMSE 0.45. Cross-validation across k confirmed $k = 15$ as the optimum." Always state how k was selected and the test-set RMSE.

13.23.8 Practical Tips

- Scale features (`scale()` or `recipes::step_normalize()`); unscaled features with different units dominate the distance computation and produce biased predictions.
- Use distance-weighted kNN (`kknn::train.kknn`) for smoother predictions and slightly better accuracy at the cost of one extra hyperparameter.
- Prediction cost is $O(nd)$ per query for naive search; use kd-trees (default in `FNN::knn.reg`) for faster lookups in low-dimensional feature spaces.
- For high-dimensional data ($d > 20$), kNN's accuracy collapses due to the curse of dimensionality; reduce dimensionality with PCA or use a parametric / tree-based model instead.
- kNN has no native handling of categorical predictors; encode as dummy variables or use Gower distance for mixed numeric-categorical inputs.
- For large datasets, consider approximate nearest-neighbour search (`RANN`, `Annoy`); exact k-NN becomes computationally prohibitive beyond a few million points.

13.23.9 R Packages Used

`FNN::knn.reg()` for fast kd-tree-based exact k-NN regression; `kknn::train.kknn()` for weighted k-NN with cross-validated kernel and k selection; `caret` and `tidymodels` for cross-validated k-NN within the broader ML workflow; `RANN` for approximate nearest-neighbour search at scale.

13.23.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.23.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.24 LightGBM

13.24.1 Introduction

LightGBM (Ke et al., 2017) is a gradient-boosting framework engineered for very large datasets. It introduces several optimisations over XGBoost: histogram-based split finding (substantially faster than sorted-value enumeration), leaf-wise tree growth (deeper trees per leaf count than level-wise), Gradient-based One-Side Sampling (GOSS) that prioritises high-gradient instances during training, and Exclusive Feature Bundling (EFB) that combines mutually exclusive sparse features. The result is typically 3–5× faster than XGBoost with comparable or slightly better accuracy on large tabular data.

13.24.2 Prerequisites

A working understanding of gradient boosting, XGBoost or similar boosted-tree workflow, and the bias-variance trade-off in tree depth and leaf count.

13.24.3 Theory

Three core innovations distinguish LightGBM from earlier gradient-boosting libraries:

- **Histogram binning**: pre-compute feature-value bins (default 255); split finding iterates over bins rather than sorted values, giving substantial speed and memory gains.
- **Leaf-wise growth**: at each iteration, expand the leaf with maximum loss reduction. Produces deeper, asymmetric trees given a fixed leaf count compared to level-wise growth, but at a higher over-fit risk.
- **GOSS**: sample instances by gradient magnitude, keeping the top-a fraction with largest gradients and a random sample of the rest. The classifier focuses on the hard examples that contribute most to the gradient.

13.24.4 Assumptions

Sufficient data for histogram bins to be informative; regularisation (`max_depth`, `num_leaves`, `min_data_in_leaf`) is tuned to prevent the over-fit risk inherent in leaf-wise growth.

13.24.5 R Implementation

```
library(lightgbm)

set.seed(2026)
n <- 600
X <- matrix(rnorm(n * 5), n, 5)
y <- as.numeric(plogis(X[, 1] + 0.5 * X[, 2]^2) > runif(n))

idx <- sample(n, n * 0.8)
d_tr <- lgb.Dataset(X[idx, ], label = y[idx])
d_te <- lgb.Dataset.create.valid(d_tr, X[-idx, ], label = y[-idx])

params <- list(objective = "binary", metric = "binary_logloss",
               learning_rate = 0.05, num_leaves = 31,
               feature_fraction = 0.8, bagging_fraction = 0.8,
               bagging_freq = 5, min_data_in_leaf = 20)

model <- lgb.train(params, d_tr,
                   nrounds = 500, valids = list(val = d_te),
                   early_stopping_rounds = 20, verbose = -1)

cat("Best iter:", model$best_iter,
    "   best logloss:", round(model$best_score, 4), "\n")

imp <- lgb.importance(model, percentage = TRUE)
head(imp)
```

13.24.6 Output & Results

`lgb.train()` returns a fitted model with the early-stopping iteration, best validation metric, and tree structure. `lgb.importance()` ranks features by gain (or split count, or cover) with optional percentage display.

13.24.7 Interpretation

A reporting sentence: "LightGBM with `learning_rate = 0.05`, `num_leaves = 31`, feature and bagging fractions 0.8 converged at 184 boosting rounds with early stopping (validation log-loss 0.39); feature 1 contributed 53 % of total gain, confirming it as the dominant predictor in line with the simulated data-generating process. Training was approximately 4× faster than XGBoost on the same data." Always state hyperparameters and early-stopping criteria.

13.24.8 Practical Tips

- Constrain `num_leaves` (default 31) or `max_depth` to prevent the over-fit risk of leaf-wise growth; deeper trees are more accurate per round but more prone to over-fitting.
- LightGBM is typically 3–5× faster than XGBoost for the same number of trees; the speed gap widens further for $>$ 1M rows.
- For categorical features, pass integer-encoded factors plus `categorical_feature` argument; LightGBM uses split-aware categorical handling that often outperforms one-hot encoding.
- A `learning_rate` of 0.01–0.05 is a sensible starting range; smaller values need more rounds but generalise better, larger values train faster.
- GPU training (`device_type = "gpu"`) provides a drop-in 3–10× acceleration for very large data; CPU performance is already excellent.
- Compare LightGBM and XGBoost on the same data; results are usually very close, but the better model varies by task.

13.24.9 R Packages Used

`lightgbm` for the canonical R interface; `xgboost` for the closely-related alternative; `catboost` for the third major gradient-boosting library (with native categorical handling); `parsnip::boost_tree(engine = "lightgbm")` for tidymodels integration; `bonsai` for the tidymodels-LightGBM bridge.

13.24.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.24.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP

- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.25 LIME Explanations

13.25.1 Introduction

LIME (Ribeiro, Singh, Guestrin 2016) explains a single prediction by fitting a sparse linear model locally around the point of interest. Samples are perturbed near the observation, labelled with the black-box model, and weighted by distance; the linear surrogate's coefficients describe the local behaviour of the model.

13.25.2 Prerequisites

Regression / classification; weighted linear regression.

13.25.3 Theory

For observation x: 1. Sample x'_k by perturbing x in feature space. 2. Predict with the black-box: $\hat{y}_k = f(x'_k)$. 3. Weight samples by proximity: $w_k = \exp(-d(x, x'_k)^2/\sigma^2)$. 4. Fit a sparse linear model g to $(x'_k, \hat{y}_k)$ with weights w_k. 5. Coefficients of g are the local explanation.

13.25.4 Assumptions

A local linear approximation captures the model's behaviour in a neighbourhood; perturbation scheme is appropriate for the feature type (Gaussian for continuous, resampling for categorical, word removal for text).

13.25.5 R Implementation

```r
library(lime); library(ranger)

set.seed(2026)
n <- 500
df <- data.frame(
  x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n),
  y  = factor(sample(c("a", "b"), n, replace = TRUE))
)
df$y <- factor(ifelse(plogis(0.7 * df$x1 - 0.4 * df$x2) > runif(n), "a", "b"))

rf <- ranger(y ~ ., data = df, num.trees = 200, probability = TRUE,
             classification = TRUE)

# LIME setup
explainer <- lime(df[1:450, -4], rf,
                  bin_continuous = TRUE, n_bins = 4)
expl      <- explain(df[451:452, -4],
                     explainer, labels = "a",
                     n_features = 3, n_permutations = 1000)

expl[, c("feature", "feature_weight", "feature_desc")]
```

13.25.6 Output & Results

Per-observation feature weights: how strongly each feature pushes the prediction toward the explained class, with a human-readable feature description.

13.25.7 Interpretation

"LIME attributed observation 1's predicted class 'a' to $x_1 > 0.8$ (weight 0.42) and $x_2 < -0.3$ (weight 0.18), providing an interpretable local rule."

13.25.8 Practical Tips

- LIME is unstable: different runs can produce different explanations due to random perturbations; set a seed and run several times.
- LIME's linear surrogate is a local approximation; for globally highly non-linear models, explanations drift across neighbourhoods.
- Compare LIME with SHAP on the same observations; large disagreements indicate explanation fragility.
- For text, LIME removes words and recomputes predictions; for tabular, use binned perturbation.
- Weighted sampling bandwidth σ strongly affects results; tune by inspection.

13.25.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.25.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.26 Linear Discriminant as ML

13.26.1 Introduction

Linear Discriminant Analysis (LDA), introduced by Ronald Fisher in 1936, is one of the oldest classification methods and remains a strong baseline. It models each class as a multivariate Gaussian with a class-specific mean but a *shared* covariance matrix, and assigns each observation to the class with the highest posterior probability under those Gaussians. The resulting decision boundary is linear in the predictors. LDA is simple, well-calibrated by construction, and often competitive with logistic regression on small to moderate-sized datasets where class-conditional Gaussianity approximately holds.

13.26.2 Prerequisites

A working understanding of multivariate Gaussian distributions, Bayes' rule for posterior class probabilities, and basic linear algebra (matrix inversion).

13.26.3 Theory

Under class-conditional multivariate Normality with shared covariance Σ and class priors π_k, the posterior is

$$P(y = k \mid x) \propto \pi_k \, \phi(x; \mu_k, \Sigma).$$

Taking the log and dropping constants common to all classes yields the linear discriminant function

$$\delta_k(x) = x^\top \Sigma^{-1} \mu_k - \tfrac{1}{2} \mu_k^\top \Sigma^{-1} \mu_k + \log \pi_k,$$

which is linear in x. The decision boundary between any two classes is thus a hyperplane. Quadratic Discriminant Analysis (QDA) relaxes the shared-covariance assumption to give class-specific Σ_k, producing quadratic boundaries at the cost of more parameters.

Fisher's geometric derivation arrives at the same linear projection by maximising the ratio of between-class to within-class scatter; this connection makes LDA both a classifier and a supervised dimensionality-reduction tool.

13.26.4 Assumptions

Class-conditional features are multivariate Normal with a shared covariance; $n > p$ for stable covariance estimation (use shrinkage LDA when p approaches n).

13.26.5 R Implementation

```r
library(MASS)

set.seed(2026)
n ← 300
x1 ← c(rnorm(n/2, 0), rnorm(n/2, 2))
x2 ← c(rnorm(n/2, 0), rnorm(n/2, 2))
y ← factor(rep(c("a", "b"), each = n/2))
df ← data.frame(x1, x2, y)

idx ← sample(n, n * 0.7)
lda_fit ← lda(y ~ ., data = df[idx, ])
pred ← predict(lda_fit, df[-idx, ])$class
mean(pred == df[-idx, "y"])

# Coefficients of the linear discriminant function
lda_fit$scaling
```

13.26.6 Output & Results

`lda()` returns the prior probabilities, class means, scaling matrix (linear discriminant coefficients), and proportion of trace each axis explains. `predict()` provides class assignments and posterior probabilities. The linear coefficients, when interpreted geometrically, point in the direction of greatest between-class separation after within-class whitening.

13.26.7 Interpretation

A reporting sentence: "LDA achieved 89 % test accuracy on the two-class Gaussian problem with shared covariance ($n_{\text{train}} = 210$, $n_{\text{test}} = 90$); the linear-discriminant direction (0.82, 0.57) corresponds to the between-class mean difference after within-class scatter whitening; logistic regression gave 87 %, indicating LDA's Gaussian assumption is well-supported." Compare LDA with logistic regression as a sanity check on the Gaussian assumption.

13.26.8 Practical Tips

- When classes are clearly Gaussian with similar covariances, LDA is nearly Bayes-optimal and outperforms logistic regression in finite samples because of its stronger model.

- For p approaching n, use shrinkage LDA (`klaR::rda()` or the `sda` package); unregularised LDA has unstable covariance estimates with poor predictions.
- LDA's posterior probabilities are well-calibrated by construction when the model holds; no Platt scaling is typically needed.
- QDA relaxes the shared-covariance assumption but needs more data per class; use it when class covariances differ visibly (Box's M test gives a formal check).
- LDA also serves as a supervised dimensionality-reduction tool: project onto the top $K - 1$ discriminants for a low-dimensional class-aware visualisation.
- For high-dimensional sparse data (text, genomics), regularised LDA (`sparseLDA`, `penalizedLDA`) handles $p \gg n$ via L1 or grouped penalties.

13.26.9 R Packages Used

`MASS::lda()` and `qda()` for the canonical implementations; `klaR::rda()` for regularised discriminant analysis; `sda` for shrinkage LDA; `penalizedLDA` for high-dimensional sparse extensions; `caret` and `tidymodels::discrim_linear()` for cross-validated LDA within ML pipelines.

13.26.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.26.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.27 Logistic Regression as ML

13.27.1 Introduction

Logistic regression models class probability as $P(y = 1 \mid x) = \sigma(w^\top x + b)$, where σ is the logistic function, and fits parameters by minimising the cross-entropy loss. As an ML workhorse it is well-calibrated by construction, interpretable through coefficients on the

log-odds scale, regularisable with L1, L2, or elastic-net penalties, and scales to millions of features — making it the standard baseline for text classification, click-through prediction, and medical risk scoring.

13.27.2 Prerequisites

A working understanding of linear models, the cross-entropy / log-loss objective, and regularisation (L1 vs L2 penalties).

13.27.3 Theory

The regularised log-loss objective is

$$L(w) = -\sum_i \left[y_i \log \sigma(w^\top x_i) + (1 - y_i) \log(1 - \sigma(w^\top x_i)) \right] + \lambda R(w),$$

with $R(w) = \|w\|_2^2$ for L2 (ridge), $\|w\|_1$ for L1 (lasso), or a convex combination for elastic net. Optimisation uses Newton-Raphson (IRLS) for the unregularised case and coordinate descent (`glmnet`) for the L1/elastic-net case. The penalty parameter λ is tuned by cross-validation, with `lambda.min` minimising CV deviance and `lambda.1se` providing a more parsimonious choice within one SE of the minimum.

13.27.4 Assumptions

Log-odds are linear in the predictors; observations are independent; sufficient sample size per feature (rule of thumb: $n > 10p$ for unregularised, less for regularised models).

13.27.5 R Implementation

```r
library(glmnet)

set.seed(2026)
n ← 400; p ← 30
X ← matrix(rnorm(n * p), n, p)
beta ← c(rep(1, 5), rep(0, 25))
y ← rbinom(n, 1, plogis(X %*% beta - 1))

# Cross-validated lasso
cv_fit ← cv.glmnet(X, y, family = "binomial",
                   alpha = 1, nfolds = 10)
plot(cv_fit)

coef_min ← as.vector(coef(cv_fit, s = "lambda.min"))
head(coef_min, 10)
cat("# nonzero:", sum(coef_min ≠ 0) - 1, "\n")
```

13.27.6 Output & Results

`cv.glmnet()` returns the cross-validation deviance curve over λ, the recommended `lambda.min` and `lambda.1se` , and the fitted coefficient paths. Lasso correctly recovered the 5 non-zero coefficients in the data-generating model; the deviance plot shows the U-shaped CV-error curve.

13.27.7 Interpretation

A reporting sentence: "Lasso logistic regression with cross-validated $\lambda_{\min} = 0.03$ retained 5 non-zero features that exactly match the data-generating β; AUC on a held-out test set was 0.84 with calibration intercept -0.02 and slope 0.97 — well-calibrated. The sparser `lambda.1se` solution retained 3 features with AUC 0.79." Always report the regularisation choice and a calibration check.

13.27.8 Practical Tips

- Standardise features before regularisation; `glmnet` does this internally and back-transforms, but explicit pre-scaling makes the workflow more transparent.
- Use `lambda.1se` for a sparser, more interpretable model; `lambda.min` is typically more accurate but less parsimonious.
- Logistic regression produces well-calibrated probabilities for correctly-specified models; SVMs, trees, and neural networks typically need Platt scaling or isotonic regression to recover calibration.
- For massively sparse data (text bags-of-words, click logs), `LiblineaR` is dramatically faster than `glmnet`; it handles millions of features routinely.
- For multi-class problems, `family = "multinomial"` in `glmnet` extends the framework natively; alternatively, `nnet::multinom()` for unregularised multinomial regression.
- For very imbalanced classes, combine logistic regression with class weights (`weights = ...`) or threshold tuning on a validation set.

13.27.9 R Packages Used

`glmnet::cv.glmnet()` for ridge, lasso, and elastic-net logistic regression; `LiblineaR` for fast sparse linear classifiers; base R `glm(family = binomial)` for unregularised logistic regression; `parsnip::logistic_reg()` for tidymodels-flavoured wrappers; `pROC` for ROC and calibration diagnostics.

13.27.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.27.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch

- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.28 A Complete mlr3 Workflow

13.28.1 Introduction

`mlr3` is the modern R6-based successor to `mlr`, a feature-complete machine-learning framework that abstracts around five core objects: Tasks (data + target + roles), Learners (algorithms), Resamplings (CV schemes), Tuners (search strategies), and Benchmarks (multi-learner comparisons). Like `tidymodels`, mlr3 provides a unified interface across dozens of underlying engines; unlike tidymodels, it uses R6 object-oriented classes rather than tibbles, giving more flexibility but a steeper learning curve.

13.28.2 Prerequisites

A working understanding of R6 classes, machine-learning pipeline concepts, and cross-validation-based hyperparameter tuning.

13.28.3 Theory

The core mlr3 workflow:

1. **Task**: wraps data with target and feature-role specifications.
2. **Learner**: an algorithm (e.g., `lrn("classif.ranger")` for random forest classification).
3. **Resampling**: a cross-validation scheme (`rsmp("cv", folds = 5)` for 5-fold CV).
4. **Tuner with AutoTuner**: wraps a learner plus a search space, inner resampling, and search strategy into a self-tuning learner.
5. **Outer resampling**: evaluate the AutoTuner on outer folds for honest nested-CV performance estimation.

`mlr3pipelines` extends the framework with preprocessing graphs and meta-learners (stacking, bagging) as composable nodes — equivalent in role to tidymodels `recipes` and `workflows`.

13.28.4 Assumptions

Task roles (target, predictors, stratification variables) are correctly specified; hyperparameter search spaces cover plausible ranges; resampling matches the inferential question (random for IID, blocked for time series, spatial for geography).

13.28.5 R Implementation

```
library(mlr3); library(mlr3learners); library(mlr3tuning)

set.seed(2026)
df ← data.frame(
  x1 = rnorm(500), x2 = rnorm(500),
  x3 = rnorm(500), x4 = rnorm(500),
  y  = factor(sample(c("yes", "no"), 500, replace = TRUE))
)

task ← as_task_classif(df, target = "y")

# Tunable random forest
lrn_rf ← lrn("classif.ranger",
             mtry = to_tune(1, 4),
             min.node.size = to_tune(5, 20),
             num.trees = 300)

at ← AutoTuner$new(
  learner    = lrn_rf,
  resampling = rsmp("cv", folds = 3),
  measure    = msr("classif.ce"),
  search_space = NULL,
  terminator = trm("evals", n_evals = 10),
  tuner      = tnr("random_search")
)

outer ← rsmp("cv", folds = 3)
rr ← resample(task, at, outer, store_models = TRUE)
rr$aggregate(msr("classif.ce"))
```

13.28.6 Output & Results

`AutoTuner` performs hyperparameter tuning on inner resampling folds and selects the best configuration. `resample()` evaluates the AutoTuner on outer folds, producing nested-CV estimates of generalisation performance. `rr$aggregate()` returns the chosen metric (classification error here) averaged across outer folds with bootstrap-based standard errors.

13.28.7 Interpretation

A reporting sentence: “Nested 3 × 3 cross-validation with mlr3 AutoTuner (10-evaluation random-search inner tuning of random forest hyperparameters) estimated classification error of 0.25 ± 0.02 for the tuned random forest; the nested wrapping prevents the optimistic bias of tuning and evaluating on the same folds.” Always state the outer/inner CV scheme and the search budget.

13.28.8 Practical Tips

- `AutoTuner` is the right pattern for honest nested cross-validation; tuning and evaluating on the same resampling produces optimistically biased estimates.
- `mlr3pipelines` formalises preprocessing as pipeline operations (PipeOps); the equivalent of tidymodels `recipes`. Compose nodes into Graphs for complex pre/post-processing.

- `mlr3tuningspaces` provides sensible default search spaces per learner; useful when you don't know the right ranges.
- `benchmark_grid()` compares multiple learners on the same task under identical resampling, giving fair head-to-head performance comparisons.
- For big data, use `mlr3spatiotempcv` or `mlr3temporal` for time-aware resamplings; standard random CV leaks information for time-series problems.
- Parallelise tuning and resampling with `future::plan(multisession)`; mlr3 respects the `future` framework.

13.28.9 R Packages Used

`mlr3` for the core framework; `mlr3learners` for the curated learner library; `mlr3tuning` and `mlr3tuningspaces` for hyperparameter optimisation; `mlr3pipelines` for graph-based pipelines; `mlr3viz` for visualisation; `mlr3spatiotempcv` and `mlr3temporal` for time- and space-aware resampling.

13.28.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.28.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.29 Naive Bayes

13.29.1 Introduction

Naive Bayes is the simplest probabilistic classifier and one of the oldest. It applies Bayes' theorem under the strong assumption that features are conditionally independent given the class label. The independence assumption is almost always wrong in practice, yet Naive Bayes performs surprisingly well — often competitively with more sophisticated methods — on text classification, spam filtering, and other high-dimensional sparse problems. It trains in linear time and serves as a strong baseline for any classification pipeline.

13.29.2 Prerequisites

A working understanding of Bayes' theorem, conditional probability, and standard distributions (Gaussian, multinomial, Bernoulli) used as per-feature class-conditional models.

13.29.3 Theory

The classifier is

$$\hat{y}(x) = \arg\max_{y} P(y) \prod_{j=1}^{p} P(x_j \mid y),$$

where $P(y)$ is the class prior (estimated from class frequencies) and $P(x_j \mid y)$ is the per-feature class-conditional density. Three standard variants:

- **Gaussian Naive Bayes**: $P(x_j \mid y)$ is Normal with class-conditional mean and variance, suitable for continuous features.
- **Multinomial Naive Bayes**: for word counts and text classification.
- **Bernoulli Naive Bayes**: for binary features.

Training reduces to computing class frequencies and per-feature class-conditional statistics — extremely fast even on millions of examples.

13.29.4 Assumptions

Features are conditionally independent given the class (almost always violated but the classifier remains useful); the chosen distribution family matches each feature's behaviour.

13.29.5 R Implementation

```
library(e1071)

set.seed(2026)
n <- 400
df <- data.frame(
  x1 = rnorm(n), x2 = rnorm(n),
  x3 = sample(c("A", "B"), n, replace = TRUE),
  y  = factor(sample(c("pos", "neg"), n, replace = TRUE))
)

# Inject a relationship
df$y <- factor(ifelse(df$x1 + 0.5 * df$x2 +
                      as.numeric(df$x3) * 0.5 > runif(n, -1, 1),
                      "pos", "neg"))

idx <- sample(n, n * 0.7)
nb <- naiveBayes(y ~ ., data = df[idx, ])
pred <- predict(nb, df[-idx, ])

mean(pred == df[-idx, "y"])
nb$tables$x1         # Gaussian parameters for x1 per class
```

13.29.6 Output & Results

`naiveBayes()` returns a fitted classifier with per-class priors and per-feature conditional distribution parameters. `predict()` provides class labels and (with `type = "raw"`) class probabilities. The accuracy on a held-out fold gives a baseline against which more complex models can be compared.

13.29.7 Interpretation

A reporting sentence: "Gaussian Naive Bayes (with Laplace smoothing for the categorical predictor) achieved 74 % test accuracy, competitive with logistic regression's 76 % at a fraction of the training cost; the per-class feature tables identify x_1 as the most discriminative feature (class-conditional mean difference 0.8)." Always state the variant used and the smoothing parameter for categorical features.

13.29.8 Practical Tips

- Naive Bayes is competitive on high-dimensional sparse data (text classification, spam filtering); often wins at small training sizes where complex models over-fit.
- Apply Laplace smoothing (`laplace = 1`) for categorical features to avoid zero-probability cases that would otherwise propagate to make the entire posterior zero.
- Probability calibration is often poor (the independence assumption produces overconfident predictions); apply Platt scaling or isotonic regression if calibrated probabilities are needed.
- For continuous features that are not Gaussian, transform (log, Box-Cox) or bin to ordinal before fitting; alternatively, use a kernel-density NB variant.
- The `naivebayes` package provides faster implementations and supports kernel density estimation for non-Gaussian continuous features.
- For text classification specifically, multinomial NB on TF-IDF features remains a strong baseline; only neural methods reliably beat it on large enough corpora.

13.29.9 R Packages Used

`e1071::naiveBayes()` for the classical implementation; `naivebayes` for faster alternatives with kernel-density and Poisson variants; `klaR::NaiveBayes()` for an interface with cross-validation hooks; `parsnip::naive_Bayes()` for tidymodels integration; `text2vec` and `quanteda` for text feature extraction before NB classification.

13.29.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.29.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.30 Nested Cross-Validation

13.30.1 Introduction

When hyperparameters are tuned on the same data used for evaluation, performance estimates are optimistically biased — the chosen hyperparameters fit not only the underlying signal but also the noise in the validation folds, producing CV errors lower than the model achieves on truly new data. Nested cross-validation separates the two concerns: an inner loop tunes hyperparameters on each training fold, and an outer loop evaluates the resulting tuned model on held-out test folds. The outer-loop mean is an honest, low-bias estimate of generalisation error.

13.30.2 Prerequisites

A working understanding of k-fold cross-validation, hyperparameter tuning, and the optimism bias that arises from tuning and evaluating on the same data.

13.30.3 Theory

The standard nested-CV scheme:

- **Outer loop**: K-fold CV. Each fold splits data into outer-training and outer-test sets.
- **Inner loop**: on each outer-training set, run K'-fold CV to pick the best hyperparameter configuration.
- **Outer evaluation**: refit the chosen configuration on the full outer-training set; predict and score on the outer-test set.
- **Aggregate**: average the K outer-test scores; the mean is the generalisation estimate, and the spread quantifies uncertainty.

Typical values: outer $K = 5$ to 10, inner $K' = 5$. Computational cost scales as $K \times K'$ times the cost of fitting one model.

13.30.4 Assumptions

Observations are iid (or the resampling respects the dependence structure for time-series, clustered, or grouped data); all preprocessing (imputation, scaling, feature engineering) is refit within each inner fold to prevent test-set leakage.

13.30.5 R Implementation

```
library(rsample); library(purrr); library(dplyr); library(tidyr)

set.seed(2026)
n <- 300
df <- tibble(x1 = rnorm(n), x2 = rnorm(n),
             y  = 1 + 2 * rnorm(n) + 0.5 * rnorm(n)^2)

outer_folds <- vfold_cv(df, v = 5)
lambdas <- c(0.001, 0.01, 0.1, 1, 10)

fit_ridge <- function(train, lambda) {
  X <- model.matrix(y ~ x1 + x2, train)[, -1]
  solve(crossprod(X) + lambda * diag(ncol(X))) %*% crossprod(X, train$y)
}
pred_ridge <- function(beta, newdata) {
  as.vector(model.matrix(y ~ x1 + x2, newdata)[, -1] %*% beta)
}

outer_rmse <- map_dbl(outer_folds$splits, function(osp) {
  tr_o <- analysis(osp); te_o <- assessment(osp)

  inner <- vfold_cv(tr_o, v = 5)
  inner_rmse <- map_dfr(lambdas, function(l) {
    errs <- map_dbl(inner$splits, function(isp) {
      b <- fit_ridge(analysis(isp), l)
      sqrt(mean((assessment(isp)$y - pred_ridge(b, assessment(isp)))^2))
    })
    tibble(lambda = l, rmse = mean(errs))
  })
  best_l <- inner_rmse$lambda[which.min(inner_rmse$rmse)]
  b <- fit_ridge(tr_o, best_l)
  sqrt(mean((te_o$y - pred_ridge(b, te_o))^2))
})

c(mean_rmse = mean(outer_rmse), sd_rmse = sd(outer_rmse))
```

13.30.6 Output & Results

A vector of K outer-fold RMSE values; the mean estimates generalisation error and the standard deviation quantifies how much that estimate varies across resampling.

13.30.7 Interpretation

A reporting sentence: “Nested 5 × 5 cross-validation estimated test RMSE of 1.04 ± 0.08 for the ridge regression with hyperparameter λ tuned on inner folds; single-CV tuning on the same hyperparameter grid would have produced RMSE 0.92 — under-estimating by

approximately 12 %, the optimism cost of tuning and evaluating on the same folds.” Always report the outer-fold mean and standard deviation, never a single CV estimate when tuning is involved.

13.30.8 Practical Tips

- Fit *all* preprocessing (imputation, scaling, feature engineering, target encoding) inside each inner fold; preprocessing on the full outer-training set leaks information into validation.
- `tidymodels::nested_cv()` and `mlr3::resampling_nested_cv()` formalise the structure and prevent the manual loop errors that nested CV is famous for.
- For small datasets ($n < 200$), repeated nested CV (repeating the outer split with different random seeds and averaging) reduces the outer-fold variance that small samples introduce.
- The final deployment model is tuned once on the *full* dataset with the chosen hyperparameter; nested CV informs uncertainty about generalisation, not the deployed model itself.
- Parallelise the inner loop with `future` or `doParallel`; the outer loop is typically fast enough to leave serial.
- Computational budget can be substantial: 5 × 5 nested CV with 5-point hyperparameter grid runs $5 \times 5 \times 5 = 125$ model fits; account for this when planning.

13.30.9 R Packages Used

`rsample` for the outer-loop splits; `tidymodels::tune` and `tune_grid()` for nested-CV-aware hyperparameter optimisation; `mlr3::resampling_nested_cv` and `AutoTuner` for mlr3-style nested CV; `caret::train` with `trainControl(method = "repeatedcv")` for an older alternative.

13.30.10 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.30.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro

- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.31 Neural Networks: Introduction

13.31.1 Introduction

Neural networks are compositions of linear maps and non-linear activation functions. A feed-forward network with L layers computes $f(x) = g_L(W_L g_{L-1}(\cdots W_2 g_1(W_1 x + b_1)\cdots) + b_L)$ where W_l are weight matrices, b_l biases, and g_l activation functions. The Universal Approximation Theorem (Cybenko, 1989) shows that even a single hidden layer with enough width can approximate any continuous function. Modern deep learning stacks hundreds of such layers to learn powerful representations for images, text, audio, and tabular data with implicit feature engineering.

13.31.2 Prerequisites

A working understanding of linear algebra (matrix multiplication), gradient descent, the chain rule for backpropagation, and basic machine-learning concepts (loss functions, train-test evaluation).

13.31.3 Theory

Perceptron: $\hat{y} = \text{sign}(w^\top x + b)$. Linear decision boundary; provably cannot solve XOR.

Multi-layer perceptron (MLP): two or more layers of linear + non-linear transformations. Universal approximator with one hidden layer of sufficient width.

Activation functions: - **Sigmoid**: $\sigma(x) = 1/(1+e^{-x})$, smooth but vanishing gradients in deep nets. - **tanh**: zero-centred sigmoid; better than sigmoid for hidden layers but still suffers vanishing gradients. - **ReLU**: $\max(0, x)$, the modern default for hidden layers; faster training and avoids vanishing gradients in positive regions. - **Leaky ReLU**, **GELU**, **SiLU**: variants that address ReLU's "dead-neuron" problem.

Training: minimise loss (MSE for regression, cross-entropy for classification) by gradient descent (or modern variants Adam, AdamW). Backpropagation computes gradients efficiently via the chain rule.

13.31.4 Assumptions

Sufficient data relative to model capacity (overparameterised models on tiny datasets memorise rather than generalise); features are normalised; loss is differentiable almost everywhere.

13.31.5 R Implementation

```r
library(nnet)

set.seed(2026)
n <- 400
x1 <- rnorm(n); x2 <- rnorm(n)
y  <- factor(ifelse(x1^2 + x2^2 > 1.5, "out", "in"))
df <- data.frame(x1 = scale(x1)[, 1], x2 = scale(x2)[, 1], y = y)

idx <- sample(n, n * 0.7)
nn <- nnet(y ~ ., data = df[idx, ],
           size = 5, decay = 1e-3, maxit = 200,
           trace = FALSE)

pred <- predict(nn, df[-idx, ], type = "class")
mean(pred == df[-idx, "y"])
```

13.31.6 Output & Results

`nnet()` fits a small MLP with one hidden layer of `size` neurons. The test accuracy on the concentric-rings task demonstrates the network's ability to learn a non-linear decision boundary that linear classifiers cannot represent.

13.31.7 Interpretation

A reporting sentence: "A 5-neuron single-hidden-layer MLP with weight decay 1e-3 trained for 200 iterations achieved 91 % test accuracy on the concentric-rings classification task; logistic regression on the same data managed only 52 %, confirming that the non-linear decision boundary requires non-linear modelling capacity." Always state architecture, regularisation, and a linear baseline.

13.31.8 Practical Tips

- Always standardise inputs; unscaled features with different units make training unstable and slow.
- Combine multiple regularisers: weight decay (L2), dropout, early stopping based on validation loss.
- Initialise weights carefully: Xavier (for tanh) or He (for ReLU) for deep networks; `nnet` uses small uniform initialisation that suffices for shallow nets.
- For tabular data, tree-based ensembles (XGBoost, LightGBM) usually outperform MLPs; neural networks dominate where structure matters (images, text, audio, time series).
- Modern R neural-network workflows use `torch` or `keras` rather than `nnet`; the latter is fine for shallow nets and pedagogy but lacks the GPU support and modern optimisers needed for production.
- For very small datasets, regularise heavily (high decay, small networks, dropout) or fall back to linear / tree models; deep nets need data to generalise.

13.31.9 R Packages Used

`nnet::nnet()` for shallow MLPs; `torch` and `luz` for modern deep learning with GPU support; `keras` and `tensorflow` as the alternative TensorFlow-based stack; `parsnip::mlp()` for tidymodels integration; `neuralnet` for an alternative implementation with visual graph rendering.

13.31.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.31.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.32 Partial Dependence Plots

13.32.1 Introduction

Partial dependence plots (PDPs), introduced by Friedman in 2001, show the average model prediction as a feature varies while holding other features at their observed values. They provide a visual answer to "how does the model's prediction change with x_j?" — exposing non-linear effects and interactions in black-box models that variable-importance scores alone cannot reveal. Combined with variable importance (which features matter) and SHAP values (per-prediction explanations), PDPs are the third pillar of interpretation for opaque ML models.

13.32.2 Prerequisites

A working understanding of trained black-box classifiers or regressors (random forest, gradient boosting, neural networks), variable importance, and the difference between marginal and conditional effects.

13.32.3 Theory

Partial dependence at value $x_j = v$ is

$$\mathrm{PD}_j(v) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}\left(v, x_{-j,i}\right),$$

i.e., set the feature of interest to v across all rows in the dataset, predict, and average. The PDP is the curve of $\mathrm{PD}_j(v)$ as v varies over the observed range of x_j.

Individual Conditional Expectation (ICE) plots show one curve per observation rather than the average; the PDP is the mean of the ICE curves. ICE plots expose heterogeneity in feature effects: if all ICE curves are roughly parallel, the effect of x_j is homogeneous; if they diverge, an interaction with another feature is implied.

Accumulated Local Effects (ALE) plots address PDP's extrapolation problem when features are correlated; they integrate the prediction's local derivative rather than averaging predictions at potentially out-of-distribution feature combinations.

13.32.4 Assumptions

For PDP interpretation, features are approximately independent; otherwise the average over x_{-j} at fixed $x_j = v$ may include low-density regions of the joint distribution and produce misleading curves. For correlated features, use ALE plots instead.

13.32.5 R Implementation

```r
library(pdp); library(ranger)

set.seed(2026)
n <- 500
df <- data.frame(
  x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n),
  y  = sin(rnorm(n)) + 0.3 * rnorm(n)^2
)

rf <- ranger(y ~ ., data = df, num.trees = 300)
pd <- partial(rf, pred.var = "x1", plot = FALSE, train = df)

plot(pd$x1, pd$yhat, type = "l", lwd = 2, col = "#2A9D8F",
     xlab = "x1", ylab = "PD(x1)",
     main = "Partial dependence of x1")
```

13.32.6 Output & Results

`pdp::partial()` returns a data frame with the feature value and the partial-dependence prediction. Plotting it as a curve shows how the average prediction varies with the feature; non-linear shapes reveal effects that linear models would miss.

13.32.7 Interpretation

A reporting sentence: "The partial-dependence plot revealed a non-monotone sinusoidal relationship between x_1 and the outcome (peak prediction near $x_1 = -0.5$, trough near $x_1 = 1.5$); this structure was not visible from variable importance alone, which simply ranked x_1 as the strongest predictor. ICE curves were broadly parallel, indicating no major interaction with other features." Always pair PDP with ICE for heterogeneity detection.

13.32.8 Practical Tips

- Pair PDP with variable importance: importance answers "which features matter?", PDP answers "how do they affect the prediction?".
- Use ICE plots to detect interactions: ICE curves diverging from the PDP indicate heterogeneous effects driven by another feature.
- PDPs extrapolate into low-density regions when features are correlated; for correlated-feature contexts, use Accumulated Local Effects (ALE) plots via `iml::FeatureEffect` or `ALEPlot`.
- Two-dimensional PDPs (pair of features) reveal interactions but are computationally expensive; use a sampled feature grid for large datasets.
- `DALEX::model_profile()` offers a unified interpretation interface across tree, linear, and deep models with consistent API.
- For per-prediction explanations rather than global ones, use SHAP values (`shapviz`, `iml`); SHAP and PDP answer related but distinct interpretation questions.

13.32.9 R Packages Used

`pdp` for the canonical PDP implementation; `iml::FeatureEffect()` for PDP, ICE, and ALE within a unified interpretation framework; `DALEX::model_profile()` for model-agnostic PDPs; `ALEPlot` for ALE specifically; `shapviz` and `fastshap` for SHAP-based per-prediction explanations.

13.32.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.32.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro

- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.33 Platt Scaling

13.33.1 Introduction

Platt scaling (Platt, 1999) is a post-hoc calibration method that maps a classifier's raw scores onto well-calibrated probabilities by fitting a logistic regression from scores to true labels on a held-out validation set. It is the simplest and often best calibration method for SVMs, neural networks, and other classifiers whose mis-calibration is approximately monotone. The idea is to leave the classifier's ranking untouched while rescaling the score axis so that the predicted probability of class 1 matches the empirical class-1 rate.

13.33.2 Prerequisites

A working understanding of calibration plots, the difference between discrimination and calibration, logistic regression, and the use of held-out validation sets.

13.33.3 Theory

Given validation pairs (f_i, y_i) where f_i is the classifier score (decision value, logit, or raw probability), fit

$$P(y = 1 \mid f) = \frac{1}{1 + \exp(Af + B)},$$

by maximum likelihood. New predictions are $P(y = 1 \mid f(x))$ from this fitted sigmoid. The transformation has only two parameters and preserves the rank order — and therefore the ROC curve and AUC — of the original classifier. Label smoothing $(y + 1)/(N + 2)$ avoids numerical issues near 0 and 1 when the validation set is small.

13.33.4 Assumptions

Mis-calibration is monotone (a sigmoid family is appropriate); a separate validation set is available for fitting the calibration; the classifier's scores rank observations meaningfully (Platt scaling cannot fix a classifier with poor discrimination).

13.33.5 R Implementation

```
set.seed(2026)
n ← 1000
# Well-ranked but mis-calibrated scores
y ← rbinom(n, 1, 0.3)
f ← 2 * y + rnorm(n, 0, 1.2)         # decision function

# Naive sigmoid of f does not match observed frequencies
naive_p ← plogis(f)

# Platt scaling: fit logistic regression y ~ f on validation
val_idx ← 1:500
te_idx  ← 501:1000
fit ← glm(y[val_idx] ~ f[val_idx], family = "binomial")
A ← coef(fit)[2]; B ← coef(fit)[1]
platt_p ← plogis(A * f[te_idx] + B)

# Brier scores
c(naive = mean((naive_p[te_idx] - y[te_idx])^2),
  platt = mean((platt_p - y[te_idx])^2))
```

13.33.6 Output & Results

Brier score (mean squared error of probability vs label) drops after Platt scaling, reflecting improved calibration. AUC remains unchanged because the rescaling is monotone and preserves rank.

13.33.7 Interpretation

A reporting sentence: “Platt scaling reduced the Brier score from 0.21 (naive sigmoid of SVM decision values) to 0.18 (Platt-rescaled) on the held-out test set; ROC AUC was unchanged at 0.81 because the rescaling preserves rank order. The reliability diagram shifted toward the diagonal at every probability decile after recalibration.” Always report Brier score and a reliability diagram before and after calibration.

13.33.8 Practical Tips

- Platt scaling works best when mis-calibration is smooth and monotone; for wiggly or non-monotone calibration curves, isotonic regression is preferred.
- Use a separate validation set (not training data) to fit the calibration; in-sample calibration is self-reinforcing and biased.
- Platt scaling adds only 2 parameters and is extremely data-efficient — a few hundred validation points are usually sufficient.
- For multi-class problems, fit one-vs-rest Platt scaling per class and renormalise the resulting probabilities to sum to 1.
- Pair with a reliability diagram (`ggcalibrate` or hand-built) before and after calibration to verify the visual improvement.
- Platt scaling cannot improve discrimination; if AUC is poor, fix the underlying classifier rather than chasing better calibration.

13.33.9 R Packages Used

Base R `glm(family = "binomial")` for fitting the Platt sigmoid; `caret::calibrate()` for an integrated calibration workflow; `probably` for tidymodels-flavoured calibration utilities; `betacal` for Beta calibration as an alternative parametric form; `mlr3calibration` for calibration extensions in the mlr3 ecosystem.

13.33.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.33.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.34 Random Forests

13.34.1 Introduction

Random forests, introduced by Breiman in 2001, average predictions from many bootstrapped decision trees, each trained with a random subset of features considered at every split. Two sources of randomness — bootstrap row sampling and feature subsampling — decorrelate the trees and dramatically reduce variance compared with a single tree. The result is a strong off-the-shelf baseline for tabular data: minimal hyperparameter tuning, robust to outliers and irrelevant features, and providing free out-of-bag (OOB) error estimation as a side effect of the bootstrap.

13.34.2 Prerequisites

A working understanding of decision trees, bagging (bootstrap aggregating), and the bias-variance trade-off in ensemble methods.

13.34.3 Theory

For each of B trees:

1. Draw a bootstrap sample (sampling with replacement, same size as original) from the training data.
2. Grow a deep tree on this sample; at each split, consider a random subset of m features (typically $m = \sqrt{p}$ for classification, $m = p/3$ for regression).
3. Predict on each tree; aggregate predictions by averaging (regression) or majority voting (classification).

Out-of-bag (OOB) error uses observations not selected by each bootstrap as a held-out validation set; aggregating these per-tree predictions across the forest gives a nearly unbiased generalisation estimate at zero additional computational cost.

13.34.4 Assumptions

Observations are independent and identically distributed; features are informative (random forests are robust to irrelevant features but cannot synthesise signal that is not there); the default m value is a starting point but may need tuning for specific datasets.

13.34.5 R Implementation

```r
library(ranger)

set.seed(2026)
n <- 500
df <- data.frame(x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n),
                 y  = factor(sample(c("pos", "neg"), n, replace = TRUE)))
df$y <- factor(ifelse(plogis(0.8 * df$x1 - 0.6 * df$x2) > runif(n), "pos", "neg"))

rf <- ranger(y ~ ., data = df,
             num.trees = 500,
             mtry = 1,
             importance = "impurity",
             classification = TRUE)

rf$prediction.error         # OOB misclassification error
rf$variable.importance
```

13.34.6 Output & Results

`ranger()` returns a fitted random forest with OOB error (`prediction.error`), per-feature variable importance (`variable.importance`), and the underlying tree structures. The OOB error is the standard generalisation estimate; variable importance ranks features by their average contribution to prediction quality.

13.34.7 Interpretation

A reporting sentence: "A 500-tree random forest with `mtry = 1` achieved OOB classification error of 0.17; variable importance identified x_1 (impurity decrease 32) and x_2 (decrease 18) as the top predictors, consistent with the data-generating logistic process. A held-out test set confirmed the OOB-based estimate (test error 0.16)." Always report OOB error or held-out test error and the top variable importances.

13.34.8 Practical Tips

- Use `ranger` instead of the original `randomForest` package for new analyses; it is dramatically faster and lower-memory while producing essentially identical results.
- Tune `mtry` on 5-fold cross-validation; the default ($\sqrt{p}$ for classification, $p/3$ for regression) is reasonable but not always optimal.
- `min.node.size` controls leaf size; larger values give more bias and less variance — useful for very noisy datasets.
- For unbiased variable importance, use `importance = "permutation"` instead of `"impurity"`; it is slower but less biased toward continuous features and high-cardinality categoricals.
- OOB error is nearly unbiased for moderate to large n; use it instead of cross-validation for rapid hyperparameter screening, then validate the final choice on a held-out test set.
- For class imbalance, use `class.weights` or `case.weights` (in `ranger`); both are natively supported and more efficient than over- or under-sampling.

13.34.9 R Packages Used

`ranger` for fast random forests with OOB and permutation importance; `randomForest` for the original Breiman implementation; `parsnip::rand_forest()` for the tidymodels-flavoured wrapper; `caret` for cross-validated tuning; `vip` for variable-importance visualisation; `pdp` for partial-dependence plots.

13.34.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.34.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.35 Preprocessing with recipes

13.35.1 Introduction

The `recipes` package in tidymodels defines preprocessing pipelines as a sequence of named steps — impute missing values, normalise numeric predictors, encode categoricals, apply PCA — that learn their parameters from training data and apply them consistently at prediction time. The principal benefit is leak-free preprocessing: imputation means, scaling parameters, and PCA loadings are estimated only on the training set, never on test or validation folds. Applying preprocessing globally before splitting (a frequent mistake) leaks information from test into training and produces optimistically biased performance estimates.

13.35.2 Prerequisites

A working understanding of tidymodels, train-test splits, cross-validation, and the concept of data leakage.

13.35.3 Theory

A recipe has two phases:

1. **Define**: chain `recipe()` with `step_*()` functions in pipeline order.
2. **Prep / bake**: `prep()` learns parameters from training data; `bake()` applies them to any new data.

All statistics (centring means, scaling SDs, imputation values, PCA loadings) come from the data passed to `prep()`. When the recipe is then applied to test data via `bake()` or via a `workflow()`, the same parameters are used — preserving the principle that test data must not influence preprocessing.

Steps are executed in declared order; ordering matters substantively (impute before scale, not after; encode categoricals before any operation that requires numeric inputs).

13.35.4 Assumptions

Training data is representative enough to estimate transform parameters reliably; step ordering is correct for the chosen transformations.

13.35.5 R Implementation

```r
library(recipes); library(dplyr)

set.seed(2026)
n <- 200
df <- tibble(
  num1 = rnorm(n),
  num2 = rnorm(n),
  cat  = sample(c("a", "b", "c"), n, replace = TRUE),
  y    = rnorm(n)
)
df$num1[sample(n, 10)] <- NA           # introduce missingness

rec <- recipe(y ~ ., data = df) %>%
  step_impute_mean(num1) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_dummy(all_nominal_predictors())

prepped <- prep(rec, training = df[1:150, ])
test_baked <- bake(prepped, new_data = df[151:200, ])
head(test_baked)
```

13.35.6 Output & Results

`prep()` returns a fitted recipe with all learnt parameters stored. `bake()` applies the prepared recipe to new data, returning a transformed tibble with imputed, normalised, and one-hot-encoded predictors. The transformations on the held-out 50 rows use parameters learned only from the first 150.

13.35.7 Interpretation

A reporting sentence: "The recipe imputed missing values in `num1` with the training-set mean (-0.02), centred and scaled all numeric predictors, and one-hot encoded the `cat` factor with three levels; all parameters were learned on the 150-row training set and applied identically to the 50-row test set, avoiding the data leakage that occurs when preprocessing is applied globally before splitting." Always describe the preprocessing steps in the methods section.

13.35.8 Practical Tips

- Place imputation steps before scaling; you cannot meaningfully standardise a column that contains NAs.
- `step_normalize()` centres and scales; `step_range()` rescales to $[0, 1]$ via min-max.
- For tree-based models (random forest, XGBoost), skip normalisation — trees are scale-invariant — but keep imputation and one-hot encoding.
- Use `step_zv()` and `step_nzv()` to remove zero-variance and near-zero-variance predictors before modelling; they contribute no information and can destabilise some algorithms.
- Pair a recipe with a model specification via `workflow()` so preprocessing and prediction happen as one pipeline; this prevents accidental leakage at predict time.

- For PCA dimensionality reduction, `step_pca()` learns loadings on training data; the same loadings are applied to test data, preserving train-test independence.

13.35.9 R Packages Used

`recipes` for the core preprocessing framework; `workflows` for pairing recipes with models; `embed` for embedding steps (target encoding, UMAP); `themis` for class-imbalance steps (SMOTE, ROSE); `textrecipes` for text-specific preprocessing; `tidymodels` as the umbrella package that bundles all of these.

13.35.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.35.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.36 Regularisation in ML

13.36.1 Introduction

Regularisation adds a complexity penalty to the training loss, preventing overfitting by shrinking coefficients. Ridge (L2) shrinks all coefficients toward zero; lasso (L1) also performs variable selection; elastic net combines both. All three are standard tools across regression and classification.

13.36.2 Prerequisites

Linear / logistic regression; convex optimisation.

13.36.3 Theory

Objective: $\min_\beta \sum_i L(y_i, x_i^\top \beta) + \lambda \cdot P(\beta)$, with - $P(\beta) = \|\beta\|^2$ (ridge, L2) - $P(\beta) = \|\beta\|_1$ (lasso, L1) - $P(\beta) = \alpha\|\beta\|_1 + (1-\alpha)\|\beta\|^2/2$ (elastic net)

Lasso produces sparse solutions (many $\hat{\beta}_j = 0$); ridge shrinks but rarely zeros; elastic net balances sparsity with the grouping effect for correlated features.

13.36.4 Assumptions

Features are standardised so the penalty is comparable across predictors; λ is tuned by CV.

13.36.5 R Implementation

```r
library(glmnet)

set.seed(2026)
n ← 300; p ← 50
X ← matrix(rnorm(n * p), n, p)
beta ← c(rep(2, 5), rep(0, 45))
y ← X %*% beta + rnorm(n)

cv_ridge ← cv.glmnet(X, y, alpha = 0,   nfolds = 10)   # L2
cv_lasso ← cv.glmnet(X, y, alpha = 1,   nfolds = 10)   # L1
cv_enet  ← cv.glmnet(X, y, alpha = 0.5, nfolds = 10)

nz ← function(fit) sum(as.vector(coef(fit, s = "lambda.min")) ≠ 0) - 1
cat("nonzero: ridge=", nz(cv_ridge),
    "  lasso=", nz(cv_lasso),
    "  elastic=", nz(cv_enet), "\n")
c(ridge = cv_ridge$cvm[cv_ridge$lambda == cv_ridge$lambda.min],
  lasso = cv_lasso$cvm[cv_lasso$lambda == cv_lasso$lambda.min],
  enet  = cv_enet $cvm[cv_enet $lambda == cv_enet $lambda.min])
```

13.36.6 Output & Results

Non-zero counts: ridge retains all 50, lasso selects ~5, elastic net selects ~7. Elastic net combines sparsity (lasso-like) with stability (ridge-like) for correlated features.

13.36.7 Interpretation

“Lasso with CV-selected λ retained the 5 true non-zero features and zero-ed out 45 irrelevant ones; elastic net gave similar sparsity but handled correlated predictors more gracefully.”

13.36.8 Practical Tips

- Always standardise features (`glmnet` does this internally); otherwise the penalty is scale-dependent.
- Use `lambda.1se` for a sparser, more parsimonious model within one SE of the CV minimum.
- Lasso picks one from correlated-feature groups arbitrarily; elastic net picks all – often more stable.
- For classification, extend via `family = "binomial"`; same rules apply.
- Gradient-boosting frameworks expose L1/L2 regularisation on leaf weights; not identical to glmnet’s lasso/ridge but same motivation.

13.36.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.36.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.37 RNNs and LSTMs: Introduction

13.37.1 Introduction

Recurrent Neural Networks (RNNs) process sequences by maintaining a hidden state that updates at every time step. Long Short-Term Memory units (LSTM; Hochreiter & Schmidhuber, 1997) add gating mechanisms to combat vanishing gradients, enabling long-range dependencies. GRUs (Cho, 2014) are a simpler gated variant.

13.37.2 Prerequisites

Feedforward networks; gradient-based training; sequence tasks.

13.37.3 Theory

Vanilla RNN: $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Gradients through time vanish or explode, limiting long-range dependence.

LSTM adds input, forget, and output gates, plus a memory cell:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \quad h_t = o_t \odot \tanh(c_t).$$

Gates control what is written, read, and forgotten in the memory cell.

Bidirectional RNNs concatenate forward and backward hidden states – useful for sequence classification.

13.37.4 Assumptions

Inputs are sequential and order-dependent; padding/masking handles variable-length sequences correctly; sufficient context in the hidden state.

13.37.5 R Implementation

```
library(torch)

torch_manual_seed(2026)

# Batch of 4 sequences, each length 10 and feature dim 5
x ← torch_randn(4, 10, 5)

lstm ← nn_lstm(input_size = 5, hidden_size = 8, num_layers = 2,
               batch_first = TRUE)
out  ← lstm(x)             # list: output tensor, (h_n, c_n)

out[[1]]$shape             # [4, 10, 8]
out[[2]][[1]]$shape        # h_n: [2, 4, 8]

# Simple forecasting model: LSTM + linear head
forecaster ← nn_module(
  initialize = function() {
    self$lstm ← nn_lstm(1, 16, batch_first = TRUE)
    self$fc   ← nn_linear(16, 1)
  },
  forward = function(x) {
    out ← self$lstm(x)[[1]]
    self$fc(out[, -1, ])       # predict from last time step
  }
)

model ← forecaster()
y_seq ← torch_randn(4, 10, 1)
model(y_seq)$shape         # [4, 1]
```

13.37.6 Output & Results

The LSTM returns per-timestep hidden states (shape [batch, seq, hidden]) plus final hidden and cell states.

13.37.7 Interpretation

“A 2-layer LSTM with 8 hidden units processed 10-step sequences; the final hidden state feeds a linear classifier. Transformers now exceed LSTMs on most NLP tasks, but LSTMs remain strong for short-window forecasting and low-data regimes.”

13.37.8 Practical Tips

- Use LSTMs or GRUs over vanilla RNNs; the latter are hard to train for any real-world sequence length.
- Pack sequences (`nn_utils_rnn_pack_padded_sequence`) to skip padding computations.

- Gradient clipping (`nn_utils_clip_grad_norm_`) is almost always needed for RNN training stability.
- For short tasks, a 1-D CNN over the sequence often matches or beats an LSTM with faster training.
- For long sequences with global dependencies, transformers beat LSTMs; but they need more data.

13.37.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.37.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.38 SHAP Values

13.38.1 Introduction

SHAP (SHapley Additive exPlanations; Lundberg & Lee, 2017) attributes each prediction to input features via the Shapley value from cooperative game theory. It is the only attribution method satisfying the combined axioms of local accuracy, consistency, and missingness. SHAP has become the standard for black-box model interpretability.

13.38.2 Prerequisites

Variable importance; cooperative game theory basics helpful.

13.38.3 Theory

For prediction $f(x)$, each feature j gets a SHAP value ϕ_j satisfying

$$f(x) = \mathbb{E}[f(X)] + \sum_j \phi_j.$$

Shapley values average the marginal contribution of feature j across all possible subsets of features, producing a fair attribution. Exact computation is exponential; Tree SHAP (Lundberg, 2020) computes it in polynomial time for tree ensembles.

13.38.4 Assumptions

Features have sensible "absence" interpretations (TreeSHAP integrates over feature distributions); additive decomposition meaningfully captures model behaviour.

13.38.5 R Implementation

```r
library(xgboost); library(shapviz)

set.seed(2026)
n <- 500
X <- matrix(rnorm(n * 4), n, 4); colnames(X) <- paste0("x", 1:4)
y <- as.numeric(X[, 1] > 0) + 0.5 * as.numeric(X[, 2] > 0) +
     rnorm(n, 0, 0.5)

dtrain <- xgb.DMatrix(X, label = y)
model <- xgb.train(list(objective = "reg:squarederror",
                        max_depth = 3, eta = 0.1),
                   data = dtrain, nrounds = 200, verbose = 0)

sv <- shapviz(model, X_pred = X, X = X)

# Summary (importance ranking)
sv_importance(sv)

# Per-observation waterfall (first prediction)
sv_waterfall(sv, row_id = 1)
```

13.38.6 Output & Results

An importance plot (mean |SHAP| per feature) and a waterfall plot showing how each feature pushes the specific prediction above or below the baseline.

13.38.7 Interpretation

"SHAP ranked x_1 as the dominant driver (mean |SHAP| = 0.36); the waterfall showed observation 1's prediction was pushed up by 0.42 from baseline primarily due to $x_1 = 1.8$."

13.38.8 Practical Tips

- Use **Tree SHAP** (fast, exact) for tree models; fallback to **Kernel SHAP** for arbitrary models (approximate, slower).
- SHAP dependence plots (SHAP value vs feature value) reveal non-linear effects and interactions.
- SHAP values sum to $f(x) - \mathbb{E}[f(X)]$, so per-observation explanations are additive.
- For deep learning, use `DeepSHAP` or integrated gradients; TreeSHAP does not apply.
- Global importance = mean |SHAP| across observations; usually close to permutation importance on tree ensembles.

13.38.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.38.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.39 SMOTE for Imbalanced Classes

13.39.1 Introduction

On imbalanced classification problems, classifiers trained with standard loss optimise for the majority class and under-predict the minority. SMOTE (Chawla et al., 2002) addresses this by synthesising new minority examples via interpolation between minority neighbours, balancing the training set without duplicating samples.

13.39.2 Prerequisites

Classification; understanding of imbalanced data; kNN.

13.39.3 Theory

SMOTE algorithm: 1. For each minority sample x_i, find its k nearest minority neighbours. 2. Select a random neighbour x_{nn}. 3. Generate a synthetic sample $x_{\text{new}} = x_i + \lambda(x_{nn} - x_i)$, $\lambda \sim U(0, 1)$.

Apply within the training set only (never the test set). Variants: Borderline-SMOTE, ADASYN (focus on hard minority regions).

13.39.4 Assumptions

Feature space is continuous (SMOTE works poorly for categoricals without adaptation); minority class has enough samples to define meaningful neighbourhoods.

13.39.5 R Implementation

```r
library(themis); library(recipes); library(dplyr)

set.seed(2026)
n <- 500
df <- tibble(
  x1 = rnorm(n), x2 = rnorm(n),
  y  = factor(sample(c("pos", "neg"), n, replace = TRUE,
                     prob = c(0.05, 0.95)))
)
table(df$y)

rec <- recipe(y ~ ., data = df) %>%
  step_smote(y, over_ratio = 1, neighbors = 5)

baked <- prep(rec) %>% bake(new_data = NULL)
table(baked$y)
```

13.39.6 Output & Results

Original class counts (~475 neg, 25 pos); after SMOTE the minority is upsampled via synthetic examples to match the majority.

13.39.7 Interpretation

“SMOTE upsampled the 5 % minority class to 50 % synthetic examples, enabling the downstream logistic regression to learn a better boundary. Test AUC improved from 0.74 to 0.83 on the held-out set.”

13.39.8 Practical Tips

- Apply SMOTE **only** to the training set (inside the CV fold), never to test/validation.
- `over_ratio = 1` creates a fully balanced training set; smaller ratios under-sample less.
- SMOTE works on numeric features; for mixed data, use SMOTE-NC or other mixed-type variants.
- Alternatives: class weighting in the loss function is often as effective and simpler.
- Evaluate using class-balanced metrics (balanced accuracy, F1, AUC, PR-AUC) – plain accuracy misleads under imbalance.

13.39.9 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.39.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.40 Stacking Ensembles

13.40.1 Introduction

Stacking, also called stacked generalisation (Wolpert 1992), combines predictions from several diverse base learners using a second-level meta-learner trained to predict the target from the base-model predictions. Stacking consistently wins competitive ML benchmarks when base models are individually decent and sufficiently diverse — the meta-learner captures complementary strengths that no single model can match. It generalises bagging (one type of model averaged) and boosting (one type of model sequentially trained) to arbitrary heterogeneous ensembles.

13.40.2 Prerequisites

A working understanding of cross-validation, the bias-variance trade-off in ensembling, and the difference between in-sample and out-of-fold predictions.

13.40.3 Theory

The standard stacking workflow:

1. **Train base learners** $M_1, M_2, \ldots, M_K$ on the training data; compute *out-of-fold* (OOF) predictions $\hat{y}_k^{\text{oof}}$ via cross-validation.
2. **Train a meta-learner** g (typically linear or regularised regression / logistic regression) to predict the target from the OOF predictions: $\hat{y} = g(\hat{y}_1^{\text{oof}}, \ldots, \hat{y}_K^{\text{oof}})$.
3. **Refit base learners** on the full training set; produce final predictions as $\hat{y}(x) = g(M_1(x), \ldots, M_K(x))$.

Using OOF predictions for the meta-learner prevents leakage that would occur if the meta-learner trained on in-sample base predictions.

13.40.4 Assumptions

Base learners are sufficiently diverse (a trees + linear + kNN combination usually outperforms five different tree variants); the meta-learner is simple enough that it does not over-fit the small set of base predictions.

13.40.5 R Implementation

```
library(tidymodels); library(stacks)

set.seed(2026)
n ← 500
df ← tibble(x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n),
            y  = factor(sample(c("pos", "neg"), n,
                               replace = TRUE, prob = c(0.3, 0.7))))

split ← initial_split(df, strata = y)
train ← training(split); test ← testing(split)

folds ← vfold_cv(train, v = 5)
ctrl ← control_stack_grid()

rec ← recipe(y ~ ., data = train)
rf_spec  ← rand_forest(mtry = 2, trees = 300) %>%
  set_engine("ranger") %>% set_mode("classification")
log_spec ← logistic_reg(penalty = 0.01) %>%
  set_engine("glmnet") %>% set_mode("classification")

rf_wf  ← workflow() %>% add_recipe(rec) %>% add_model(rf_spec)
log_wf ← workflow() %>% add_recipe(rec) %>% add_model(log_spec)

rf_res  ← fit_resamples(rf_wf,  folds, control = ctrl)
log_res ← fit_resamples(log_wf, folds, control = ctrl)

st ← stacks() %>% add_candidates(rf_res) %>% add_candidates(log_res)
st_blend ← blend_predictions(st)
st_fit   ← fit_members(st_blend)

st_fit
```

13.40.6 Output & Results

`fit_members()` returns the trained stack with meta-learner coefficients weighting each base model. Inspecting `st_blend` shows the chosen blending weights — typically a non-negative least-squares or penalised regression solution that emphasises the base models with the most distinctive predictive signal.

13.40.7 Interpretation

A reporting sentence: “Stacking a random forest and a regularised logistic regression via a penalised linear meta-learner achieved test ROC AUC of 0.89, exceeding either base model alone (RF 0.85, logistic 0.80); the meta-learner weighted RF at 0.62 and logistic at 0.38, with both contributions retained at non-zero values.” Always report base-model performances, the meta-learner choice, and the blending weights.

13.40.8 Practical Tips

- Base-model diversity (tree + linear + kNN + neural net) matters more than having many similar learners; a stack of five XGBoost variants is rarely better than a single well-tuned XGBoost.

- Use a regularised linear meta-learner (lasso or non-negative LS); unregularised meta-learners over-fit the relatively small set of OOF predictions.
- Pass *out-of-fold* predictions to the meta-learner, not in-sample predictions; this is the central correctness guarantee of stacking.
- Stacking plus bagging hybrids (super learner of van der Laan, Polley, Hubbard) perform best on moderate-to-large data.
- The `stacks` package (tidymodels) automates the workflow and integrates with the broader `tune` ecosystem for cross-validated base-model tuning.
- `SuperLearner` is the classical alternative outside tidymodels; both produce equivalent results given the same base models.

13.40.9 R Packages Used

`stacks` for tidymodels-based stacking; `SuperLearner` for the classical implementation; `tune::fit_resamples()` for the cross-validated base-learner predictions; `parsnip` for unified base-model specifications; `glmnet` and `ranger` for the standard base-learner choices.

13.40.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.40.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.41 Supervised Learning: Overview

13.41.1 Introduction

Supervised learning uses labelled data $(x_i, y_i)_{i=1}^n$ to learn a function f that predicts y from new x. The enterprise is defined by three choices: a **hypothesis class** (tree, linear model, neural net), a **loss function** (squared error, cross-entropy), and an **optimisation** procedure. Generalisation – performance on unseen data – is the goal, and the bias-variance decomposition is the classical lens for understanding why models fail.

13.41.2 Prerequisites

Probability, linear algebra, basic statistics.

13.41.3 Theory

Empirical risk minimisation (ERM): $\hat{f} = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_i L(y_i, f(x_i))$.

Loss functions: - Regression: squared loss $(y - f)^2$, absolute loss $|y - f|$. - Classification: 0/1 (intractable), log loss (cross-entropy), hinge (SVMs).

Bias-variance decomposition for squared loss:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \text{Bias}^2 + \text{Variance} + \sigma^2_{\text{noise}}.$$

Flexible models have low bias, high variance; rigid models the reverse. Regularisation trades one for the other.

13.41.4 Assumptions

(x, y) pairs are drawn iid from some joint distribution; the deployment distribution matches the training distribution (no distribution shift).

13.41.5 R Implementation

```
library(tidymodels)

set.seed(2026)
n ← 300
x ← runif(n, 0, 2 * pi)
y ← sin(x) + rnorm(n, 0, 0.3)
df ← tibble(x = x, y = y)

split ← initial_split(df, prop = 0.7)
train ← training(split); test ← testing(split)

# Two models of different flexibility
m_rigid ← lm(y ~ x, data = train)
m_flex  ← loess(y ~ x, data = train, span = 0.2)

rmse ← function(yhat, y) sqrt(mean((yhat - y)^2))
rmse(predict(m_rigid, test), test$y)
rmse(predict(m_flex,  test), test$y)
```

13.41.6 Output & Results

The flexible loess model has lower test RMSE on this non-linear signal; the rigid linear model under-fits.

13.41.7 Interpretation

“The sinusoidal signal is captured by loess (test RMSE 0.31) but missed by linear regression (test RMSE 0.68), illustrating how bias dominates when the hypothesis class is too restrictive.”

13.41.8 Practical Tips

- Always report test performance, never training performance, as a generalisation estimate.
- The no-free-lunch theorem: no single model wins everywhere; match model to the data-generating process.
- Stratify classification splits to preserve class proportions.
- Use cross-validation for small-to-moderate data; a single train/test split is high-variance.
- Track multiple metrics: RMSE and MAE together are more informative than either alone.

13.41.9 For Reviewers

Use the shared reviewer checklist, together with this method’s assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript’s application of the method.

13.41.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.42 Kernel SVMs

13.42.1 Introduction

Kernel SVMs extend the linear SVM framework to non-linear decision boundaries via the kernel trick: replace every inner product $x_i^\top x_j$ in the dual formulation with a kernel function $K(x_i, x_j)$ that corresponds to an inner product in a high-dimensional (often

infinite-dimensional) implicit feature space. The Radial Basis Function (RBF) kernel is the workhorse, capable of representing essentially any smooth decision boundary given enough training data and appropriate hyperparameter tuning.

13.42.2 Prerequisites

A working understanding of linear SVMs, inner-product spaces, the dual formulation of constrained optimisation, and Mercer's theorem (the conditions a function must satisfy to be a valid kernel).

13.42.3 Theory

Common kernels:

- **RBF (Gaussian)**: $K(x, x') = \exp(-\gamma \|x - x'\|^2)$. Universal approximator; the default choice for most non-linear problems.
- **Polynomial**: $K(x, x') = (x^\top x' + c)^d$. Useful when polynomial structure is expected from domain knowledge.
- **Linear**: $K(x, x') = x^\top x'$. Reduces to the standard linear SVM.
- **Sigmoid**: $K(x, x') = \tanh(\alpha x^\top x' + c)$, related to single-layer perceptrons.

Two hyperparameters dominate: the cost parameter C (margin tightness, inherited from linear SVM) and a kernel-specific parameter (γ for RBF, d and c for polynomial). RBF γ trades bias for variance: small γ produces a smoother decision boundary, large γ produces a wigglier one with high variance.

13.42.4 Assumptions

Features are appropriately scaled; sufficient data to support the kernel's implicit dimensionality (RBF in particular grows in effective complexity with γ); C and γ are tuned jointly because they are correlated.

13.42.5 R Implementation

```r
library(e1071)

set.seed(2026)
n <- 300
theta <- runif(n, 0, 2 * pi)
r <- c(rep(1, n/2), rep(2.5, n/2)) + rnorm(n, 0, 0.3)
df <- data.frame(x1 = r * cos(theta), x2 = r * sin(theta),
                 y  = factor(rep(c("inner", "outer"), each = n/2)))
df[, 1:2] <- scale(df[, 1:2])

svm_rbf <- svm(y ~ ., data = df,
               kernel = "radial", cost = 1, gamma = 0.5,
               scale = FALSE)
table(predict(svm_rbf, df), df$y)

# Grid-search C x gamma via tune()
tune_out <- tune(svm, y ~ ., data = df, kernel = "radial",
                 ranges = list(cost = c(0.1, 1, 10),
                               gamma = c(0.1, 0.5, 1)))
tune_out$best.parameters
```

13.42.6 Output & Results

The RBF-kernel SVM separates the two concentric classes — a non-linear pattern that no linear classifier can solve. `tune()` performs grid-search cross-validation over C and γ and returns the best combination by CV accuracy.

13.42.7 Interpretation

A reporting sentence: "The RBF-kernel SVM correctly separated the concentric-rings classes with grid-CV-selected $C = 1$ and $\gamma = 0.5$ (CV accuracy 0.96, test accuracy 0.94); the linear-kernel baseline managed only 0.51, confirming the non-linear structure cannot be captured without the kernel trick." Always state both hyperparameters and how they were tuned.

13.42.8 Practical Tips

- Always standardise features; kernel distances are scale-sensitive and unscaled features bias the geometry.
- For RBF, γ trades bias (small γ, smoother boundary) for variance (large γ, wigglier boundary); tune jointly with C on a logarithmic 2D grid.
- C and γ are correlated; larger C paired with smaller γ often gives similar performance to smaller C with larger γ.
- High-degree polynomial kernels are rarely useful in practice; RBF subsumes most cases and is more numerically stable.
- For large n ($> 100{,}000$), kernel SVMs become computationally expensive ($O(n^2)$ to $O(n^3)$ training); consider Nyström approximation (`kernlab::kha`) or fall back to primal linear SVM on engineered features.
- For probability output, set `probability = TRUE` in `e1071::svm()`; it fits a Platt-scaling layer on top of the SVM decision values.

13.42.9 R Packages Used

`e1071::svm()` for the canonical kernel SVM with RBF, polynomial, and sigmoid options; `kernlab::ksvm()` for an alternative with richer kernel customisation; `LiblineaR` for the primal linear-only fast variant; `parsnip::svm_rbf()` and `svm_poly()` for tidymodels integration; `caret` for cross-validated joint tuning.

13.42.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.42.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.43 Linear Support Vector Machines

13.43.1 Introduction

Linear support vector machines (SVMs) find the hyperplane that maximally separates classes with the largest possible margin. The soft-margin formulation introduced by Cortes and Vapnik in 1995 added slack variables to tolerate misclassified or margin-violating points, governed by a regularisation parameter C. Linear SVMs scale to very large sparse feature spaces (common in text classification, where bag-of-words gives millions of features) and are competitive with logistic regression on most tabular classification tasks.

13.43.2 Prerequisites

A working understanding of linear classification, convex optimisation, and the geometric concept of margin in supervised learning.

13.43.3 Theory

The soft-margin primal optimisation problem is

$$\min_{w,b,\xi} \tfrac{1}{2}\|w\|^2 + C\sum_i \xi_i \quad \text{s.t.} \quad y_i(w^\top x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0.$$

The first term maximises the margin (minimising $\|w\|$); the second penalises slack. Larger C penalises slack more, favouring low training error at the cost of higher variance; smaller C allows more margin violations and produces a smoother boundary. The dual formulation exposes the support-vector structure: only points on or inside the margin contribute to w, so the trained model is effectively determined by a sparse subset of training points.

13.43.4 Assumptions

Classes are approximately linearly separable for the chosen C; features are appropriately scaled (SVMs are distance-based and unscaled features dominate the geometry).

13.43.5 R Implementation

```
library(e1071)

set.seed(2026)
n ← 200
x1 ← rnorm(n); x2 ← rnorm(n)
y ← factor(ifelse(x1 + x2 + rnorm(n, 0, 0.5) > 0, "pos", "neg"))
df ← data.frame(x1, x2, y)

# Scale the features
df[, 1:2] ← scale(df[, 1:2])

svm_fit ← svm(y ~ ., data = df,
              kernel = "linear",
              cost = 1,
              scale = FALSE)

table(pred = predict(svm_fit, df), truth = df$y)

# Vary C and observe support-vector count
cs ← c(0.01, 0.1, 1, 10, 100)
sapply(cs, function(c) {
  f ← svm(y ~ ., data = df, kernel = "linear", cost = c, scale = FALSE)
  c(C = c, n_sv = f$tot.nSV)
})
```

13.43.6 Output & Results

`svm()` returns a fitted classifier with the support vectors, dual coefficients, and a learned weight vector. Varying C shows the trade-off: small C retains many support vectors (soft boundary), large C retains few (tight boundary, high variance).

13.43.7 Interpretation

A reporting sentence: "A linear SVM with cross-validated $C = 1$ achieved 92 % test accuracy with 124 support vectors out of 200 training points; the support-vector count indicates the classes overlap moderately and a soft boundary is appropriate. Higher C

values reduced support-vector counts but increased test error, confirming the bias-variance trade-off." Always report C, support-vector counts, and how C was tuned.

13.43.8 Practical Tips

- Standardise features before fitting; unscaled features with different units bias the geometry of the margin.
- Tune C by cross-validation over a logarithmic grid (typical range 10^{-3} to 10^{3}); the U-shaped CV-error curve has a clear optimum.
- For class imbalance, use `class.weights` to up-weight the minority class; alternatively, balance via under-sampling.
- For very large sparse feature spaces (text classification with millions of features), `LiblineaR` is dramatically faster than `e1071::svm` and equally accurate.
- Linear SVM and logistic regression often produce nearly identical decision boundaries on the same data; the SVM uses hinge loss while logistic uses log-loss, but for most tasks the choice is convention rather than substance.
- For non-linearly separable data, switch to kernel SVM (RBF, polynomial); the linear SVM cannot represent non-linear boundaries regardless of C.

13.43.9 R Packages Used

`e1071::svm()` for the canonical SVM implementation; `LiblineaR` for fast linear SVMs on large sparse data; `kernlab::ksvm()` for an alternative with richer kernel support; `parsnip::svm_linear()` for tidymodels integration; `caret` for cross-validated C tuning.

13.43.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.43.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.44 A Complete tidymodels Workflow

13.44.1 Introduction

`tidymodels` is the R meta-package that wraps `rsample`, `recipes`, `parsnip`, `workflows`, `tune`, and `yardstick` into a consistent, tidyverse-friendly interface for machine-learning pipelines. A complete tidymodels workflow composes a data split, a preprocessing recipe, a model specification, hyperparameter tuning on cross-validation resamples, and final evaluation on a held-out test set. The framework's strength is that the same syntax handles regression, classification, and time-series tasks across dozens of underlying engines (lm, glm, random forest, xgboost, neural networks).

13.44.2 Prerequisites

A working knowledge of R basics, the tidyverse, and standard model-fitting concepts (training-test split, cross-validation, hyperparameter tuning).

13.44.3 Theory

The canonical tidymodels pipeline:

1. **Split data** with `rsample::initial_split()` into training and test sets.
2. **Define preprocessing** with `recipes`: imputation, normalisation, dummy encoding, feature engineering.
3. **Specify a model** with `parsnip`, choosing an engine and which hyperparameters to tune.
4. **Combine** into a `workflow` that bundles recipe and model so preprocessing is applied consistently at prediction time.
5. **Tune** hyperparameters on cross-validation resamples (`tune::tune_grid` or Bayesian `tune_bayes`) using a metric set (`yardstick`).
6. **Finalise** on the best hyperparameters and refit on all training data.
7. **Evaluate** on the test set, never touching it before this step.

13.44.4 Assumptions

Data are tidy (one row per observation, one column per variable); roles (outcome vs predictor) are clearly identified; cross-validation splits are appropriate to the problem (random for IID, stratified for imbalance, blocked for time series).

13.44.5 R Implementation

```r
library(tidymodels)

set.seed(2026)
df <- tibble(
  x1 = rnorm(500), x2 = rnorm(500),
  x3 = rnorm(500), x4 = rnorm(500),
  y  = factor(sample(c("yes", "no"), 500, replace = TRUE,
                     prob = c(0.3, 0.7)))
)

split <- initial_split(df, strata = y)
train <- training(split); test <- testing(split)
folds <- vfold_cv(train, v = 5, strata = y)

rec <- recipe(y ~ ., data = train) %>%
  step_normalize(all_numeric_predictors())

spec <- rand_forest(trees = 300, mtry = tune(), min_n = tune()) %>%
  set_engine("ranger") %>% set_mode("classification")

wf <- workflow() %>% add_recipe(rec) %>% add_model(spec)

grid <- expand_grid(mtry = c(1, 2, 3), min_n = c(5, 15))
tuned <- tune_grid(wf, folds, grid = grid, metrics = metric_set(roc_auc))

best <- select_best(tuned, metric = "roc_auc")
final_fit <- finalize_workflow(wf, best) %>% last_fit(split)

collect_metrics(final_fit)
```

13.44.6 Output & Results

`tune_grid()` returns a tibble with one row per combination of resample fold and hyperparameter setting, including the metric value (here ROC AUC). `last_fit()` refits the workflow on the full training set with the chosen hyperparameters and evaluates it on the held-out test set in one call.

13.44.7 Interpretation

A reporting sentence: “tidymodels random-forest workflow with 5-fold stratified cross-validation selected `mtry = 2`, `min_n = 5` (best CV ROC AUC 0.81 across the 6-point grid); on the held-out test set the final model achieved ROC AUC 0.79, the honest generalisation estimate.” Always state the resampling scheme, the tuning metric, and the test-set performance.

13.44.8 Practical Tips

- Use `workflow()` rather than direct `fit()` calls; the workflow ensures preprocessing is applied consistently at prediction time without leaking test-set information into transformations.
- `last_fit()` refits on the full training set and evaluates on the test set in a single command; this is the cleanest pattern for the final-model step.

- Stratify splits and CV folds by outcome (`strata = y`) for imbalanced classification; otherwise rare classes can be missing in some folds.
- `hardhat::extract_workflow()`, `extract_fit_engine()`, and `extract_recipe()` retrieve internals from a fitted workflow for inspection or downstream use.
- Parallelise tuning with `doParallel::registerDoParallel(cores = 8)` or the `future` framework; tuning is often embarrassingly parallel across resamples and hyperparameter combinations.
- For Bayesian hyperparameter optimisation, use `tune_bayes()` instead of `tune_grid()`; it converges to the optimum in fewer evaluations on smooth performance landscapes.

13.44.9 R Packages Used

`tidymodels` as the umbrella package; `rsample` for splits and resampling; `recipes` for preprocessing; `parsnip` for unified model specifications; `workflows` for bundling; `tune` for hyperparameter optimisation; `yardstick` for performance metrics; `dials` for hyperparameter ranges; `finetune` for racing-based tuning that prunes poor candidates early.

13.44.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.44.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.45 Train-Test Splits

13.45.1 Introduction

A train-test split reserves a random subset of the data for honest evaluation of model generalisation, separating it cleanly from the data used for fitting and tuning. It is the simplest possible evaluation protocol and the foundation of every supervised-learning

workflow. The trade-off is straightforward: a larger test set gives more stable error estimates but leaves less data for fitting; for small datasets, cross-validation extracts more value from the available data than a single fixed split.

13.45.2 Prerequisites

A working understanding of supervised learning, the train-validate-test paradigm, and the difference between hyperparameter tuning and final model evaluation.

13.45.3 Theory

With n observations, reserve n_{test} for test. Typical ratios are 80/20, 70/30, or 90/10 depending on dataset size. The test-set error is a point estimate of generalisation with variance $O(1/n_{\text{test}})$ — for stable error estimates the test set should be at least a few hundred observations.

Stratified splits sample within each class so class proportions match between train and test; this is essential for imbalanced classification, where random sampling can leave the minority class severely under- or over-represented in either fold.

Grouped splits keep all observations from the same subject, site, or sample together to prevent leakage; ignoring grouping when it exists silently inflates performance estimates.

13.45.4 Assumptions

Observation pairs (x_i, y_i) are independent and identically distributed (or the dependence structure is respected by the chosen split scheme); deployment distribution matches the training distribution.

13.45.5 R Implementation

```
library(rsample); library(dplyr)

set.seed(2026)
n ← 1000
df ← tibble(x = rnorm(n),
            y = factor(sample(c("pos", "neg"), n,
                              replace = TRUE, prob = c(0.1, 0.9))))

# Random 80/20 split
split1 ← initial_split(df, prop = 0.8)

# Stratified split preserving y proportions
split2 ← initial_split(df, prop = 0.8, strata = y)

tbl_noStrata ← table(training(split1)$y) / nrow(training(split1))
tbl_strata   ← table(training(split2)$y) / nrow(training(split2))

round(rbind(noStrata = tbl_noStrata, strata = tbl_strata), 3)
```

13.45.6 Output & Results

The proportion table shows that stratified splitting preserves the 10 % minority-class proportion in both train and test; random splitting can drift by 2–3 percentage points, especially with smaller test sets. `training(split)` and `testing(split)` extract the two folds.

13.45.7 Interpretation

A reporting sentence: "An 80/20 stratified split preserved the 10 % minority-class proportion in both training and test folds (training: 10.1 %, test: 9.9 %); we tuned hyperparameters on the training set via 5-fold cross-validation and reserved the test set for a single final evaluation, never touching it during model development." Always state the split ratio, stratification, and how the test set was used.

13.45.8 Practical Tips

- For time series, split chronologically (`rsample::initial_time_split()`); random splitting leaks future information into training and produces optimistic estimates.
- For grouped data (multiple observations per subject), use `rsample::group_initial_split()` to prevent leakage; ignoring grouping when it exists silently inflates performance estimates.
- Test sets should be large enough for stable error estimates; below 200 observations the test error has high variance and a single split may not generalise.
- The test set is touched *once*: after model development is complete. Repeated use during development implicitly tunes against it and produces optimism that no single number can capture.
- For final deployment, refit the best model on the training and test sets combined; the test set served its purpose at evaluation time and there is no reason to discard its information.
- For small datasets, prefer cross-validation over a single train-test split; CV uses the data more efficiently and gives a less variable error estimate.

13.45.9 R Packages Used

`rsample::initial_split()` and `initial_time_split()` for splits; `group_initial_split()` and `group_vfold_cv()` for grouped data; `caret::createDataPartition()` for an older alternative; `mlr3::partition()` for mlr3-style splits; base R `sample()` for ad-hoc splitting.

13.45.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.45.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.46 Transformers: Overview

13.46.1 Introduction

The Transformer architecture (Vaswani et al., 2017) replaced recurrent networks as the dominant sequence model. Its core mechanism is **self-attention**: each token attends to every other token in the sequence with learned weights. Transformers underpin BERT, GPT, T5, and modern LLMs, and have expanded into vision (ViT), audio (wav2vec2), and tabular domains.

13.46.2 Prerequisites

Neural networks; attention concept; tokenisation basics.

13.46.3 Theory

Scaled dot-product attention:

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where Q, K, V are query, key, and value projections of the input.

Multi-head attention runs h parallel attention heads and concatenates outputs.

Transformer block = multi-head self-attention + feedforward + residual + layer norm, stacked many times.

Positional encoding injects sequence order (sinusoidal or learned).

13.46.4 Assumptions

Sufficient data (transformers are data-hungry); proper tokenisation (BPE or WordPiece for text); positional encoding matches task.

13.46.5 R Implementation

```
library(torch)

torch_manual_seed(2026)

# Minimal transformer block: multi-head attention + feedforward
block ← nn_module(
  initialize = function(d_model = 32, n_heads = 4, d_ff = 64) {
    self$mha ← nn_multihead_attention(embed_dim = d_model,
                                      num_heads = n_heads,
                                      batch_first = TRUE)
    self$ln1 ← nn_layer_norm(d_model)
    self$ff  ← nn_sequential(nn_linear(d_model, d_ff),
                             nn_relu(),
                             nn_linear(d_ff, d_model))
    self$ln2 ← nn_layer_norm(d_model)
  },
  forward = function(x) {
    attn ← self$mha(x, x, x)[[1]]
    x ← self$ln1(x + attn)
    x ← self$ln2(x + self$ff(x))
    x
  }
)

model ← block()
seq ← torch_randn(2, 10, 32)        # batch=2, seq=10, d_model=32
out ← model(seq)
out$shape                            # [2, 10, 32]
```

13.46.6 Output & Results

Output sequence of the same shape as the input; the block transforms representations by letting each token attend to the full context.

13.46.7 Interpretation

“A single transformer block with 4-head self-attention and feedforward sublayers transformed a (10-token, 32-dim) sequence. Stacking 6-48 such blocks gives modern NLP models; pretrained backbones (BERT, RoBERTa) are the practical starting point.”

13.46.8 Practical Tips

- Transformers need a lot of data; for small-data tasks, fine-tune a pretrained backbone rather than training from scratch.
- Use efficient attention variants (FlashAttention, sparse attention) for long sequences.
- Masking (causal for GPT, padding for BERT) is essential; forgetting it silently breaks training.
- Positional encoding: sinusoidal for classical transformers, rotary (RoPE) for modern LLMs.
- For R, practical transformer pipelines often call HuggingFace via `reticulate`; `torch` in R implements building blocks but lacks a huge pretrained-model ecosystem.

13.46.9 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.46.10 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.47 Variable Importance

13.47.1 Introduction

Variable importance ranks features by contribution to model accuracy. It is the most common first-pass interpretability tool for black-box models. Tree ensembles offer two native measures: impurity-based (mean decrease in Gini or variance) and permutation-based (drop in accuracy when the feature is shuffled).

13.47.2 Prerequisites

Decision trees; random forests; cross-validation.

13.47.3 Theory

Impurity importance: sum the impurity decrease over all splits using feature j. Biased toward high-cardinality features and correlated features.

Permutation importance: shuffle feature j in held-out data; measure the drop in performance. Unbiased for correlated features as a group but can split importance between correlated pairs.

Conditional permutation and SHAP values handle correlation more rigorously (at higher cost).

13.47.4 Assumptions

Sufficient held-out data for permutation; features correctly treated as continuous or categorical.

13.47.5 R Implementation

```r
library(ranger); library(vip)

set.seed(2026)
n <- 500
df <- data.frame(
  x1 = rnorm(n), x2 = rnorm(n), x3 = rnorm(n),
  x4 = rnorm(n), x5 = rnorm(n),
  y  = sin(rnorm(n)) + 0.3 * rnorm(n)^2
)

rf <- ranger(y ~ ., data = df, num.trees = 500,
             importance = "permutation")
imp <- sort(rf$variable.importance, decreasing = TRUE)
print(round(imp, 3))

vip(rf, num_features = 5) +
  ggplot2::labs(title = "Permutation importance (ranger RF)")
```

13.47.6 Output & Results

Sorted importance scores; the top feature dominates, others are close to zero on this mostly-noise data.

13.47.7 Interpretation

“Permutation importance identified x_1 as the dominant predictor (12 % RMSE increase when shuffled); other features contributed negligibly, consistent with the data-generating process.”

13.47.8 Practical Tips

- Use permutation importance when correctness matters; impurity importance is cheap but biased.
- Drop correlated features or use grouped importance if groups of correlated variables share the signal.
- Importance alone does not reveal direction of effect; pair with partial dependence or SHAP.
- Normalise importance by the baseline model error to express as a relative increase.
- For deep learning, use integrated gradients or SHAP; RF-style permutation is not standard.

13.47.9 Reporting

When reporting variable importance, always state which flavour was used (impurity, permutation, drop-column, or SHAP), the loss against which importance is measured, and whether the held-out test set or the training data drove the calculation. Permutation importance computed on training data overstates the apparent role of variables that the model overfits to, so the test-set version is the more defensible default. In a manuscript, include a small bar chart or table for the top features and note explicitly that importance is a measure of model reliance, not of causal effect or biological mechanism. Where features are correlated, mention that shared signal can split importance across columns and that grouped-permutation summaries may be more interpretable.

13.47.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.47.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+
- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.48 XGBoost

13.48.1 Introduction

XGBoost (eXtreme Gradient Boosting), introduced by Chen and Guestrin in 2016, extends classical gradient boosting with L1 and L2 regularisation of leaf weights, a second-order Taylor expansion of the loss, sparsity-aware split finding, and histogram-based methods that scale to billions of rows. It has dominated competitive machine-learning benchmarks on structured tabular data for nearly a decade and remains the strongest single off-the-shelf model for most regression and classification problems on tables.

13.48.2 Prerequisites

A working understanding of gradient boosting, decision trees, regularisation (L1/L2 penalties), and cross-validation for hyperparameter tuning.

13.48.3 Theory

The XGBoost objective is

$$\mathcal{L} = \sum_i L(y_i, \hat{y}_i) + \sum_k \Omega(f_k), \qquad \Omega(f) = \gamma T + \tfrac{1}{2}\lambda \|w\|^2,$$

where L is the per-observation loss (squared error, log-loss, etc.), f_k is the k-th regression tree, T is the number of leaves, w are the leaf weights, γ penalises tree complexity, and λ regularises leaf weights. A second-order Taylor expansion of L yields analytic optimal leaf weights and a tractable split score.

Key hyperparameters: `eta` (learning rate), `max_depth`, `min_child_weight`, `lambda` and `alpha` (L2 and L1 leaf-weight regularisation), `gamma` (minimum loss reduction for a split), `subsample` (row sampling), `colsample_bytree` (column sampling at tree level).

13.48.4 Assumptions

Dataset is tabular; labels suit the chosen objective (binary, multiclass, count, regression); sufficient sample size to support tree-based interactions and depth.

13.48.5 R Implementation

```
library(xgboost)

set.seed(2026)
n ← 600
X ← matrix(rnorm(n * 5), n, 5)
y ← as.numeric(plogis(X[, 1] + 0.5 * X[, 2]^2) > runif(n))

idx ← sample(n, n * 0.8)
d_tr ← xgb.DMatrix(X[idx, ], label = y[idx])
d_te ← xgb.DMatrix(X[-idx, ], label = y[-idx])

params ← list(objective = "binary:logistic", eta = 0.05,
              max_depth = 4, subsample = 0.8,
              colsample_bytree = 0.8,
              eval_metric = "logloss")

bst ← xgb.train(params = params, data = d_tr,
                nrounds = 500,
                watchlist = list(val = d_te),
                early_stopping_rounds = 20,
                verbose = 0)

cat("Best iter:", bst$best_iteration,
    "  best logloss:", round(bst$best_score, 4), "\n")

xgb.importance(model = bst)[1:5, ]
```

13.48.6 Output & Results

`xgb.train()` returns a fitted `xgb.Booster` with the early-stopping iteration count, best validation metric, and tree structure. `xgb.importance()` ranks features by gain (the average loss reduction attributed to splits on that feature), cover, or frequency.

13.48.7 Interpretation

A reporting sentence: "XGBoost with `eta = 0.05`, `max_depth = 4`, row and column subsampling at 0.8, converged after 217 boosting rounds with early stopping at 20-round patience; validation log-loss 0.38, ROC AUC 0.92. Feature-gain importance identified x_1 and x_2 as the strongest contributors, consistent with the simulated data-generating process." Always state the hyperparameters, early-stopping settings, and validation metrics.

13.48.8 Practical Tips

- Always use early stopping (`early_stopping_rounds`) to avoid over-training; without it, XGBoost trains until `nrounds` regardless of whether the validation loss has plateaued.
- Choose `objective` to match the target: `"reg:squarederror"` for regression, `"binary:logistic"` for binary classification, `"multi:softprob"` for multiclass, `"count:poisson"` for counts, `"survival:cox"` for time-to-event.
- Tune with Bayesian optimisation (`ParBayesianOptimization`, `tune::tune_bayes()`) rather than grid search; XGBoost has many correlated knobs and grid search wastes evaluations.
- `tree_method = "hist"` accelerates training 5–10× on large data; `"gpu_hist"` adds another order of magnitude on GPU.
- Monotone constraints (`monotone_constraints`) enforce domain-knowledge directions for predictor-outcome relationships; useful in regulated industries (credit risk, healthcare).
- For interpretation beyond feature importance, use SHAP values (`xgboost::predict(..., predcontrib = TRUE)` or the `shapviz` package); SHAP gives per-prediction explanations.

13.48.9 R Packages Used

`xgboost` for the canonical R interface; `parsnip::boost_tree()` for the tidymodels wrapper; `lightgbm` and `catboost` as competitive alternatives; `shapviz` and `iml` for SHAP-based interpretation; `ParBayesianOptimization` for efficient hyperparameter tuning.

13.48.10 For Reviewers

Use the shared reviewer checklist, together with this method's assumptions and practical tips.

Method-specific reviewer notes have not yet been developed for this entry. The links above support a structured review; they do not certify the example or a manuscript's application of the method.

13.48.11 See also — labs in this chapter

- Cross-validation, nested CV, bootstrap .632+

- Regularisation: ridge, lasso, elastic net
- Trees, random forests, gradient boosting
- Interpretability and SHAP
- Tabular neural networks with torch
- Imaging and sequence models intro
- tidymodels pipelines
- FDR, knockoffs, replication crisis

13.49 Week 1, Session 1 — Cross-validation, nested CV, bootstrap .632+

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.49.1 Learning objectives

- Implement k-fold and repeated k-fold cross-validation from scratch and with a package, and report an honest generalisation error.
- Explain why selecting a model *and* tuning a hyperparameter on the same fold inflates performance, and use nested CV to fix it.
- Compute a bootstrap .632+ estimate and say when it is preferable to CV.

13.49.2 Prerequisites

Sampling variability; comfort with `caret` or base-R model fitting.

13.49.3 Background

Every predictive model reports an error on the data it was fit to. That number is almost always optimistic, because the fit has had a chance to memorise noise as well as signal. Cross-validation and the bootstrap are two resampling strategies that simulate the experience of scoring the model on data it has not seen. They are not interchangeable: they make different trade-offs between bias and variance, and they behave differently under high-dimensional settings.

A particular failure mode that recurs in the applied literature is *double dipping*: using the same held-out fold both to choose a hyperparameter and to report the final error. The fold is then no longer held out, and the reported error slides back toward the training error. Nested cross-validation is the clean solution: the inner loop picks hyperparameters, the outer loop reports error.

The bootstrap gives a different window onto generalisation. The plain bootstrap resamples with replacement; the .632 and .632+ variants weight the in-bag and out-of-bag errors to reduce bias when the learner overfits. In very small samples the .632+ estimator is often the most stable, and it is worth carrying in your toolbox alongside CV.

A practical note: when you write up a modelling study, the generalisation error is the headline number. If you tuned, you must disclose how. If the inner tuning cost was hidden in the outer fold, a careful reader will notice that the reported error is too good and cannot be reproduced. Nested CV is not a cosmetic fix; it is the difference between a number that generalises and a number that does not.

13.49.4 Setup

```
library(tidyverse)
library(glmnet)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.49.5 1. Goal

Estimate the generalisation error of a lasso on a moderate *p*, small *n* simulated regression problem, first naively, then with a proper nested cross-validation.

13.49.6 2. Approach

We simulate `n = 120` observations with `p = 40` features, of which only 5 are true signal. We will (i) fit a single `cv.glmnet`, (ii) report the CV-minimising error, then (iii) wrap that in an outer 5-fold loop to show that the inner CV error is optimistic.

```
n ← 120; p ← 40
X ← matrix(rnorm(n * p), n, p)
beta ← c(rep(1.5, 5), rep(0, p - 5))
y ← as.numeric(X %*% beta + rnorm(n, sd = 1.0))

tibble(yhat_naive = as.numeric(X %*% beta), y = y) ▷
  ggplot(aes(yhat_naive, y)) +
  geom_point(alpha = 0.6) +
  geom_abline(slope = 1, intercept = 0, colour = "grey50") +
  labs(x = "True linear predictor", y = "Observed y")
```

13.49.7 3. Execution

Naive single-layer CV.

```
cv1 ← cv.glmnet(X, y, alpha = 1, nfolds = 5)
naive_mse ← min(cv1$cvm)
naive_mse
```

Nested CV. The inner `cv.glmnet` picks `lambda`; the outer 5 folds produce the reported MSE.

```r
K ← 5
folds ← sample(rep(1:K, length.out = n))
outer_mse ← numeric(K)
for (k in 1:K) {
  tr ← folds ≠ k; te ← folds == k
  fit_k ← cv.glmnet(X[tr, ], y[tr], alpha = 1, nfolds = 5)
  yhat  ← as.numeric(predict(fit_k, newx = X[te, ], s = "lambda.min"))
  outer_mse[k] ← mean((y[te] - yhat)^2)
}
nested_mse ← mean(outer_mse)
c(naive = naive_mse, nested = nested_mse)
```

13.49.8 4. Check

Bootstrap .632+ error as a cross-check.

```r
B ← 50
errs_in ← errs_oob ← numeric(B)
for (b in 1:B) {
  idx ← sample.int(n, n, replace = TRUE)
  oob ← setdiff(seq_len(n), unique(idx))
  fit_b ← cv.glmnet(X[idx, ], y[idx], alpha = 1, nfolds = 5)
  yhat_in  ← as.numeric(predict(fit_b, newx = X[idx, ],  s = "lambda.min"))
  yhat_oob ← as.numeric(predict(fit_b, newx = X[oob, ],  s = "lambda.min"))
  errs_in[b]  ← mean((y[idx] - yhat_in)^2)
  errs_oob[b] ← mean((y[oob] - yhat_oob)^2)
}
err_app ← mean(errs_in); err_oob ← mean(errs_oob)
err_632 ← 0.368 * err_app + 0.632 * err_oob
c(apparent = err_app, oob = err_oob, boot632 = err_632,
  nested = nested_mse)
```

13.49.9 5. Report

> On simulated data (n = 120, p = 40, five true signals), a lasso with tuning parameter chosen by 5-fold CV achieved an out-of-sample MSE of `round(nested_mse, 2)` under nested 5 × 5 CV. The naive single-layer CV understated the error at `round(naive_mse, 2)`; the bootstrap .632 estimator gave `round(err_632, 2)`, which bracketed the nested estimate.

The naive number looks tighter, but it reflects the choice of `lambda` as well as the fit. Only the nested number is a fair estimate of what this procedure — tune-then-fit — will do on new data.

13.49.10 Common pitfalls

- Tuning on the test fold and reporting the resulting error.
- Using a single 80/20 split in high dimensions; variance across splits can exceed the effect of interest.
- Bootstrapping without tracking in-bag vs out-of-bag, which inflates the estimate back to the apparent error.

13.49.11 Further reading

- Hastie, Tibshirani, Friedman, *Elements of Statistical Learning*, §7.10–7.11.
- Varma S, Simon R (2006), *Bias in error estimation when using cross-validation for model selection*.
- Efron B, Tibshirani R (1997), *Improvements on cross-validation: the .632+ bootstrap method*.

13.49.12 Session info

```
sessionInfo()
```

13.49.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.50 Week 4, Session 4 — FDR, knockoffs, replication crisis

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.50.1 Learning objectives

- Simulate *p*-values under a mixture of null and alternative hypotheses and apply Benjamini–Hochberg FDR control.
- Describe the knockoff filter in words and explain what "exchangeable" means in that context.
- Connect FDR discipline to the replication crisis in biomedicine.

13.50.2 Prerequisites

Multiple-testing basics.

13.50.3 Background

Multiple-testing corrections come in two families: family-wise error rate (Bonferroni, Holm) control the probability of *any* false discovery; false-discovery rate methods (Benjamini–Hochberg, Storey's *q*-value) control the *proportion* of false discoveries among the discoveries. FDR is the appropriate target for the typical screen with thousands of simultaneous tests and a tolerance for some false positives on the shortlist.

Knockoffs are a newer idea that replace the classical *p*-value route entirely. For each feature a "knockoff" variable is constructed that matches the feature's joint distribution with other features but is conditionally independent of the outcome. A statistic comparing the original and the knockoff then controls the FDR without distributional assumptions on the test statistic, provided the knockoffs are exchangeable.

The replication crisis in biomedicine is not usually a failure of FDR in the technical sense; it is usually a failure to control *researcher degrees of freedom* — the multiple analyses that precede the reported test. No post-hoc correction can rescue a flexible, undocumented workflow.

Pre-registration, code availability, and transparent reporting discipline the part of the pipeline that FDR cannot. They are not optional extras.

13.50.4 Setup

```
library(tidyverse)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.50.5 1. Goal

Simulate 1 000 tests, most null and a few alternative, and compare uncorrected, Bonferroni, and BH discovery sets.

13.50.6 2. Approach

```
m ← 1000
m1 ← 50
is_alt ← c(rep(TRUE, m1), rep(FALSE, m - m1))
mu ← ifelse(is_alt, 3, 0)
z ← rnorm(m, mu, 1)
p ← 2 * (1 - pnorm(abs(z)))

tibble(p = p, is_alt = is_alt) ▷
  ggplot(aes(p, fill = is_alt)) +
  geom_histogram(bins = 40, position = "identity", alpha = 0.7) +
  labs(x = "p-value", y = "count")
```

13.50.7 3. Execution

```
bonf ← p.adjust(p, method = "bonferroni")
bh   ← p.adjust(p, method = "BH")

res ← tibble(
  method = c("raw p<0.05", "Bonferroni<0.05", "BH q<0.05"),
  discovered  = c(sum(p < 0.05), sum(bonf < 0.05), sum(bh < 0.05)),
  true_pos    = c(sum(p[is_alt] < 0.05),
                  sum(bonf[is_alt] < 0.05),
                  sum(bh[is_alt] < 0.05)),
  false_pos   = c(sum(p[!is_alt] < 0.05),
                  sum(bonf[!is_alt] < 0.05),
                  sum(bh[!is_alt] < 0.05))
) ▷ mutate(fdp = false_pos / pmax(discovered, 1))
res
```

13.50.8 4. Check

Knockoff discussion — the idea, in words.

```r
library(knockoff)
# For a linear regression y ~ X with X gaussian, construct
# Xk matching the second-order structure of X but conditionally
# independent of y given X. Then fit a penalised regression on
# [X, Xk]; the per-feature statistic W_j = |beta_j| - |beta_{j+p}|
# controls the FDR among selected features.
ko ← knockoff.filter(X, y, fdr = 0.1)
ko$selected
```

13.50.9 5. Report

On 1 000 simulated tests (50 alternative, 950 null), raw *p*-value cut-off at 0.05 produced `res$discovered[1]` discoveries with a false-discovery proportion of `round(res$fdp[1], 2)`. Bonferroni correction produced `res$discovered[2]` discoveries; the BH procedure at $q < 0.05$ produced `res$discovered[3]` with a false-discovery proportion of `round(res$fdp[3], 2)`.

Knockoffs would give a similar FDR guarantee via a different route, and would remain valid under correlation between features provided the joint distribution can be modelled.

13.50.10 Common pitfalls

- Reporting nominal $p < 0.05$ cut-offs in a high-dimensional screen.
- Applying BH to a set of tests that are not approximately independent or positive-dependent.
- Treating knockoffs as a drop-in for *p*-values without checking the exchangeability assumption.

13.50.11 Further reading

- Benjamini Y, Hochberg Y (1995), *Controlling the false discovery rate: A practical and powerful approach to multiple testing*.
- Barber RF, Candès EJ (2015), *Controlling the false discovery rate via knockoffs*.
- Ioannidis JPA (2005), *Why most published research findings are false*.

13.50.12 Session info

```r
sessionInfo()
```

13.50.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.51 Week 2, Session 4 — Imaging and sequence models

intro

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.51.1 Learning objectives

- Describe how a 2-D convolution extracts spatial features from an image.
- Describe how a 1-D convolution or a recurrent layer processes a sequence.
- Recognise when to use a CNN, RNN, or transformer for biomedical data.

13.51.2 Prerequisites

A neural-network training loop in `torch`.

13.51.3 Background

Convolutional neural networks (CNNs) exploit the spatial locality of image data: a small filter is slid across the image, producing a feature map for each filter. Stacking convolutional layers lets the network learn increasingly abstract features, from edges to textures to whole tissue structures. In biomedical imaging — histopathology, radiology, dermatology — CNNs underpin the majority of deep-learning applications.

For sequence data — gene sequences, electronic health record time series, wearable signals — the relevant operations are 1-D convolutions and attention. Transformers have become the dominant architecture for long sequences, but 1-D CNNs and gated recurrent networks remain competitive on shorter inputs and are easier to train. This lab illustrates the *shape* of the computation with a tiny hand-built 2-D convolution and a tiny 1-D convolution; the real training happens with frameworks that we mark `#| eval: false`.

The conceptual point: image and sequence models share the training tabular `torch` training loop. What differs is the architecture, the data pipeline, and the inductive biases (locality for CNNs, ordered context for RNNs and attention).

13.51.4 Setup

```
library(tidyverse)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.51.5 1. Goal

Show manually computed convolutions on a small image and sequence, then sketch the `torch` code that would perform the same operations inside a trained model.

13.51.6 2. Approach

Create a toy 16×16 image with a bright square.

```
img ← matrix(0, 16, 16)
img[5:10, 5:10] ← 1
img ← img + matrix(rnorm(256, 0, 0.1), 16, 16)

image(t(img[16:1, ]), col = grey.colors(100), main = "toy image")
```

13.51.7 3. Execution

A manual 3×3 edge filter convolved with the image.

```
kernel ← matrix(c(-1, -1, -1, -1, 8, -1, -1, -1, -1), 3, 3)
conv2d ← function(img, k) {
  kh ← nrow(k); kw ← ncol(k)
  out ← matrix(0, nrow(img) - kh + 1, ncol(img) - kw + 1)
  for (i in seq_len(nrow(out))) for (j in seq_len(ncol(out)))
    out[i, j] ← sum(img[i:(i + kh - 1), j:(j + kw - 1)] * k)
  out
}
feat ← conv2d(img, kernel)
image(t(feat[nrow(feat):1, ]), col = grey.colors(100),
      main = "edge-filter feature map")
```

A 1-D convolution on a toy sequence.

```
seq1 ← c(rep(0, 10), rep(1, 5), rep(0, 10))
conv1d ← function(x, k) {
  out ← numeric(length(x) - length(k) + 1)
  for (i in seq_along(out)) out[i] ← sum(x[i:(i + length(k) - 1)] * k)
  out
}
feat1d ← conv1d(seq1, c(-1, 1))
tibble(i = seq_along(feat1d), v = feat1d) ▷
  ggplot(aes(i, v)) + geom_line() +
  labs(x = "position", y = "edge response")
```

A tiny CNN in `torch` (sketch only).

```
library(torch)
cnn ← nn_sequential(
  nn_conv2d(1, 8, kernel_size = 3, padding = 1),
  nn_relu(),
  nn_max_pool2d(2),
  nn_conv2d(8, 16, kernel_size = 3, padding = 1),
  nn_relu(),
  nn_max_pool2d(2),
  nn_flatten(),
  nn_linear(16 * 4 * 4, 10)
)
```

A 1-D convolutional sequence model sketch.

```
library(torch)
seq_model <- nn_sequential(
  nn_conv1d(1, 8, kernel_size = 5, padding = 2),
  nn_relu(),
  nn_adaptive_avg_pool1d(1),
  nn_flatten(),
  nn_linear(8, 1)
)
```

13.51.8 4. Check

Inspect the convolution output shape and magnitude.

```
dim(feat)
summary(as.numeric(feat))
```

13.51.9 5. Report

> A 3×3 edge filter applied to a 16×16 toy image produced a feature map of shape `paste(dim(feat), collapse = "×")`, with responses concentrated at the boundary of the bright square. Analogous 1-D convolutions applied to a binary step sequence produce sharp derivative-like responses at the transitions.

Real CNNs and sequence models compose many such filters, learned from data. The lab establishes the building block so the architectures in the literature read as stacks of familiar operations, not black boxes.

13.51.10 Common pitfalls

- Applying a CNN to tabular data; locality inductive bias is wasted.
- Forgetting input channel and batch dimensions in `torch`.
- Comparing accuracy of sequence models without specifying the tokenisation used.

13.51.11 Further reading

- LeCun Y, Bengio Y, Hinton G (2015), *Deep learning*.
- Vaswani A et al. (2017), *Attention is all you need*.

13.51.12 Session info

```
sessionInfo()
```

13.51.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.52 Week 2, Session 2 — Interpretability and SHAP

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.52.1 Learning objectives

- Produce permutation importance, partial-dependence, and Shapley values for a fitted black-box model, and interpret each plot.
- State the limitations of feature-attribution methods under feature correlation.
- Wrap a model with `DALEX` and `iml` for a standardised explanation pipeline.

13.52.2 Prerequisites

Trees and forests.

13.52.3 Background

Interpretability is not a single method; it is a cluster of tools that answer different questions. *Permutation importance* answers: how much worse does the model score if I scramble this feature? *Partial dependence* answers: averaging over all the other features, how does the prediction change as this feature varies? *SHAP* answers: for this specific prediction, how much did each feature contribute relative to the average prediction?

All three can mislead under feature correlation. Permuting a feature breaks its correlation with others, producing unrealistic inputs. Partial dependence averages over a marginal distribution that may not represent the data. SHAP — in particular TreeSHAP for trees — has several variants (interventional vs conditional) that answer subtly different causal questions. The robust practice is to use more than one method and to check that they tell a consistent story.

Interpretability is not a substitute for a causal model. A feature can have large SHAP values and large permutation importance and still be downstream of the outcome, a proxy for a confounder, or a leakage artefact. The explanation tells you what the model uses; whether that is what the world uses is a separate, harder question.

13.52.4 Setup

```
library(tidyverse)
library(ranger)
library(DALEX)
library(iml)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.52.5 1. Goal

Fit a random forest on the `palmerpenguins` dataset and produce three complementary interpretability artefacts.

13.52.6 2. Approach

```
d ← palmerpenguins::penguins ▷
  tidyr::drop_na() ▷
  mutate(species = factor(species))
ggplot(d, aes(bill_length_mm, body_mass_g, colour = species)) +
  geom_point(alpha = 0.7)
```

13.52.7 3. Execution

```
rf ← ranger(species ~ bill_length_mm + bill_depth_mm +
              flipper_length_mm + body_mass_g,
            data = d, probability = TRUE, num.trees = 500)

explainer ← DALEX::explain(
  model = rf,
  data  = d[, c("bill_length_mm", "bill_depth_mm",
                "flipper_length_mm", "body_mass_g")],
  y     = as.integer(d$species == "Adelie"),
  predict_function = function(m, newdata) predict(m, newdata)$predictions[, "Adelie"],
  label = "rf-Adelie",
  verbose = FALSE
)
vi ← DALEX::model_parts(explainer, type = "variable_importance")
plot(vi)
```

```
pdp ← DALEX::model_profile(explainer, variables = "bill_length_mm")
plot(pdp)
```

13.52.8 4. Check

A Shapley value for one instance.

```
pred_fun ← function(model, newdata) predict(model, newdata)$predictions[, "Adelie"]
predictor ← iml::Predictor$new(rf, data = d[, 3:6],
                               y = as.integer(d$species == "Adelie"),
                               predict.function = pred_fun)
sh ← iml::Shapley$new(predictor, x.interest = d[1, 3:6])
plot(sh)
```

13.52.9 5. Report

> On the penguins dataset, a random forest predicting the probability of *Adelie* species had permutation-importance values dominated by bill length and flipper length. Partial-dependence profiles showed that Adelie probability decreases sharply above a bill length of about 44 mm. For the first instance in the dataset, the Shapley attribution confirmed bill length as the primary contributor.

Report all three artefacts together, and note the correlation structure among the features when you do.

13.52.10 Common pitfalls

- Treating SHAP values as causal effects.
- Running permutation importance on correlated features without a conditional variant.
- Presenting partial-dependence plots outside the data's support.

13.52.11 Further reading

- Lundberg SM, Lee S-I (2017), *A unified approach to interpreting model predictions*.
- Molnar C, *Interpretable Machine Learning* (online book).

13.52.12 Session info

```
sessionInfo()
```

13.52.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.53 Week 1, Session 2 — Regularisation: ridge, lasso, elastic net

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.53.1 Learning objectives

- Fit ridge, lasso, and elastic-net regressions with `glmnet`, and explain what each penalty does to the coefficient path.
- Select `lambda` and `alpha` by cross-validation, and describe the bias–variance trade-off the choice implies.
- Interpret a coefficient path figure and a set of standardised coefficients.

13.53.2 Prerequisites

OLS and GLMs; the cross-validation lab.

13.53.3 Background

The ordinary least-squares solution is unbiased but, when p is close to or larger than n, it is unstable: small perturbations of the data move the fitted coefficients a long way. Regularisation trades a little bias for a lot of variance reduction by adding a penalty to the fitting criterion. Ridge shrinks all coefficients toward zero, lasso sets some exactly to zero, and elastic net blends the two. Each lives on the `glmnet` coefficient path and is selected by the same basic mechanism: cross-validation over `lambda` (and optionally `alpha`).

In biomedical data the motivation is rarely pure prediction. A lasso that selects 7 of 400 genes is also a shortlist for the next experiment. A ridge that keeps every variable but shrinks their effects is an honest admission that signal is diffuse and that no small subset will carry the story. Elastic net is the default when variables come in correlated groups, because the lasso tends to pick one at random from a correlated cluster while ridge keeps them all.

Standardise before fitting. The penalty is scale-sensitive, and a variable in raw units will be penalised differently from the same variable in standardised units. `glmnet` standardises by default, but it is worth knowing what the coefficients refer to when you plot them.

13.53.4 Setup

```
library(tidyverse)
library(glmnet)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.53.5 1. Goal

Compare ridge, lasso, and elastic net on a simulated high-dimensional regression problem and pick a final model.

13.53.6 2. Approach

`n = 100`, `p = 200`, block-correlated design with a handful of true signals, to exercise both the sparsity of lasso and the group-handling of elastic net.

```
n ← 100; p ← 200
Z ← matrix(rnorm(n * (p / 10)), n, p / 10)
X ← Z[, rep(seq_len(p / 10), each = 10)] + 0.2 * matrix(rnorm(n * p), n, p)
beta ← c(rep(2, 5), rep(-1.5, 5), rep(0, p - 10))
y ← as.numeric(X %*% beta + rnorm(n))

tibble(i = seq_along(beta), beta = beta) ▷
  ggplot(aes(i, beta)) + geom_segment(aes(xend = i, yend = 0)) +
  labs(x = "feature index", y = "true coefficient")
```

13.53.7 3. Execution

```
fit_ridge ← glmnet(X, y, alpha = 0)
fit_lasso ← glmnet(X, y, alpha = 1)
fit_enet  ← glmnet(X, y, alpha = 0.5)
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1))
plot(fit_ridge, xvar = "lambda"); title("ridge")
plot(fit_lasso, xvar = "lambda"); title("lasso")
plot(fit_enet,  xvar = "lambda"); title("elastic net")
```

13.53.8 4. Check

Pick `lambda` by CV for each; compare minimum CV MSE.

```r
cv_ridge ← cv.glmnet(X, y, alpha = 0,   nfolds = 5)
cv_lasso ← cv.glmnet(X, y, alpha = 1,   nfolds = 5)
cv_enet  ← cv.glmnet(X, y, alpha = 0.5, nfolds = 5)
tibble(
  model = c("ridge", "lasso", "enet"),
  min_cvmse = c(min(cv_ridge$cvm), min(cv_lasso$cvm), min(cv_enet$cvm)),
  nonzero   = c(sum(coef(cv_ridge, s = "lambda.min") ≠ 0),
                sum(coef(cv_lasso, s = "lambda.min") ≠ 0),
                sum(coef(cv_enet,  s = "lambda.min") ≠ 0))
)
```

13.53.9 5. Report

> On a simulated `n = 100, p = 200` regression with 10 true signals, elastic net (α = 0.5) achieved the lowest 5-fold CV MSE (`round(min(cv_enet$cvm), 2)`) and retained `sum(coef(cv_enet, s = "lambda.min") ≠ 0) - 1` features. Lasso was comparable but dropped one member of each correlated signal pair; ridge was smoothest but produced no sparse shortlist.

The choice is problem-dependent: if the next step is validation of a candidate biomarker panel, lasso or elastic net is preferable for the interpretable sparsity; if the next step is prediction only, ridge is often more stable.

13.53.10 Common pitfalls

- Interpreting lasso coefficients as unbiased effect estimates.
- Forgetting that correlated features share shrinkage: lasso may pick any one of them.
- Reporting CV-minimum error as generalisation error (see the cross-validation lab).

13.53.11 Further reading

- Friedman J, Hastie T, Tibshirani R (2010), *Regularization paths for generalized linear models via coordinate descent*.
- Zou H, Hastie T (2005), *Regularization and variable selection via the elastic net*.

13.53.12 Session info

```r
sessionInfo()
```

13.53.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.54 Week 2, Session 3 — Tabular neural networks with torch

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.54.1 Learning objectives

- Sketch the architecture of a small feed-forward network for tabular regression using `torch`.
- Write a training loop with manual batching and an optimiser.
- Compare a neural network to a linear baseline and be honest about when the network is worth the complexity.

13.54.2 Prerequisites

Regularisation; basic gradient descent intuition.

13.54.3 Background

Neural networks rarely beat gradient-boosted trees on small tabular biomedical problems. They come into their own when the feature representation itself must be learned (images, sequences, free text) or when the sample size is large enough to feed a flexible model. On a feature matrix of hundreds of rows and tens of columns, a logistic or linear regression with appropriate regularisation is a better baseline and often a better final model.

That said, understanding the training loop — forward pass, loss, backward pass, optimiser step — is a prerequisite for everything in imaging, sequence, and modern generative modelling. This lab walks through that loop on a small simulated regression so the shapes and operations are transparent.

We mark the full training loop with `#| eval: false` because installation of `torch` downloads a library that may not be present in every rendering environment. A tiny parallel example using base R gradient descent is shown first so the ideas run end-to-end no matter what is installed.

13.54.4 Setup

```
library(tidyverse)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.54.5 1. Goal

Fit a small regression by manual gradient descent, then show the equivalent `torch` code with `#| eval: false`.

13.54.6 2. Approach

```
n ← 500; p ← 10
X ← matrix(rnorm(n * p), n, p)
beta ← c(2, -1, 0.5, rep(0, p - 3))
y ← as.numeric(X %*% beta + rnorm(n, 0, 0.5))
tibble(i = seq_len(n), y = y) ▷
  ggplot(aes(i, y)) + geom_point(alpha = 0.3, size = 0.6)
```

13.54.7 3. Execution

Base-R gradient descent on the same problem.

```
w ← rep(0, p); lr ← 0.01
losses ← numeric(200)
for (step in 1:200) {
  yhat ← X %*% w
  resid ← as.numeric(y - yhat)
  grad ← -2 * t(X) %*% resid / n
  w ← w - lr * grad
  losses[step] ← mean(resid^2)
}
tibble(step = 1:200, loss = losses) ▷
  ggplot(aes(step, loss)) + geom_line() +
  labs(x = "iteration", y = "mean squared error")
```

The equivalent `torch` code.

```
library(torch)

x_t ← torch_tensor(X, dtype = torch_float())
y_t ← torch_tensor(matrix(y, ncol = 1), dtype = torch_float())

net ← nn_sequential(
  nn_linear(p, 16), nn_relu(),
  nn_linear(16, 1)
)
optimizer ← optim_adam(net$parameters, lr = 1e-2)
loss_fn ← nn_mse_loss()

for (epoch in 1:200) {
  optimizer$zero_grad()
  pred ← net(x_t)
  loss ← loss_fn(pred, y_t)
  loss$backward()
  optimizer$step()
}
```

13.54.8 4. Check

Compare our gradient-descent weights against the OLS solution.

```
lm_coef ← as.numeric(coef(lm(y ~ X - 1)))
tibble(var = seq_along(w), gd = as.numeric(w), lm = lm_coef, truth = beta) ▷
  pivot_longer(-var) ▷
  ggplot(aes(var, value, colour = name)) +
  geom_point() + geom_line(alpha = 0.5) +
  labs(x = "feature index", y = "coefficient")
```

13.54.9 5. Report

> A small feed-forward network and a linear model both recover the true coefficients on a simulated `n = 500, p = 10` regression. Under this signal-to-noise ratio and sample size, the linear model is the appropriate baseline; the neural network is shown here for machinery, not as the recommended estimator.

The training-loop pattern — zero grads, forward, loss, backward, step — is the same whether you are fitting a 3-layer MLP on penguins or a ResNet on histology. The difference is the architecture, the data pipeline, and the compute.

13.54.10 Common pitfalls

- Fitting a deep MLP on a small tabular problem and then reporting training loss as generalisation error.
- Forgetting to standardise inputs before training; learning rates are calibrated on normalised scales.
- Ignoring reproducibility: set both the R seed and the torch seed.

13.54.11 Further reading

- Goodfellow I, Bengio Y, Courville A, *Deep Learning*, ch. 6.
- `torch for R` book by Keydana (online).

13.54.12 Session info

```
sessionInfo()
```

13.54.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.55 Week 2, Session 5 — tidymodels pipelines

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.55.1 Learning objectives

- Build a `tidymodels` recipe, workflow, and tuning grid for a binary classification task.
- Split data for CV properly and collect metrics across resamples.
- Finalise a model on the full training data and evaluate on a test split.

13.55.2 Prerequisites

Cross-validation and regularisation.

13.55.3 Background

`tidymodels` organises the modelling workflow into composable objects: a *recipe* describes preprocessing, a *model specification* describes the algorithm and its tunable parameters, a *workflow* bundles them, and *tune* functions drive resampling. The value of this separation is reproducibility: the same recipe runs on training, tuning, and test data, and it is hard to leak information across the split.

This lab builds a full pipeline on `MASS::Pima.tr`: logistic regression with elastic net and tuned `mixture` and `penalty`. The same pattern extends to random forests, boosting, neural networks, or any model with a parsnip wrapper.

The convention to remember: tune on the training folds, collect metrics, finalise with the best parameters, fit once to all of the training data, then evaluate once on the test set. The test set is touched exactly once.

13.55.4 Setup

```
library(tidyverse)
library(tidymodels)
library(MASS)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.55.5 1. Goal

Fit an elastic-net logistic regression to predict diabetes status on `Pima.tr`, using a proper tidymodels pipeline.

13.55.6 2. Approach

```
d ← as_tibble(MASS::Pima.tr)
ggplot(d, aes(glu, bmi, colour = type)) + geom_point(alpha = 0.7)
```

13.55.7 3. Execution

Split, recipe, model, workflow, tune.

```
d_split ← initial_split(d, prop = 0.75, strata = type)
d_train ← training(d_split); d_test ← testing(d_split)
folds ← vfold_cv(d_train, v = 5, strata = type)

rec ← recipe(type ~ ., data = d_train) ▷
  step_normalize(all_numeric_predictors())

mod ← logistic_reg(penalty = tune(), mixture = tune()) ▷
  set_engine("glmnet")

wf  ← workflow() ▷ add_recipe(rec) ▷ add_model(mod)

grid ← grid_regular(penalty(), mixture(), levels = 5)
res  ← tune_grid(wf, resamples = folds, grid = grid,
                 metrics = metric_set(roc_auc, accuracy))
collect_metrics(res) ▷
  filter(.metric == "roc_auc") ▷
  arrange(desc(mean)) ▷
  head(5)
```

13.55.8 4. Check

Select best, finalise, evaluate on the test set.

```
best ← select_best(res, metric = "roc_auc")
wf_final ← finalize_workflow(wf, best)
fit_final ← fit(wf_final, data = d_train)
preds ← predict(fit_final, d_test, type = "prob") ▷
  bind_cols(predict(fit_final, d_test), d_test)
roc_auc(preds, truth = type, .pred_Yes, event_level = "second")
accuracy(preds, truth = type, estimate = .pred_class)
```

```
roc_curve(preds, truth = type, .pred_Yes, event_level = "second") ▷
  autoplot()
```

13.55.9 5. Report

An elastic-net logistic regression tuned on 5-fold CV, using a 5×5 grid over `penalty` and `mixture`, achieved a test ROC-AUC of `round(roc_auc(preds, truth = type, .pred_Yes, event_level = "second")$.estimate, 3)` on `MASS::Pima.tr`.

The pipeline pattern — recipe + workflow + tune + finalise — is the template we will reuse for every predictive-model lab hereafter.

13.55.10 Common pitfalls

- Normalising the entire dataset before splitting; leakage is then baked in.
- Reporting the mean CV metric as if it were a test-set metric.
- Tuning without stratifying folds on an imbalanced outcome.

13.55.11 Further reading

- Kuhn M, Silge J, *Tidy Modeling with R* (online book).

13.55.12 Session info

```
sessionInfo()
```

13.55.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)

13.56 Week 2, Session 1 — Trees, random forests, gradient boosting

Workflow labs use the variant template: Goal → Approach → Execution → Check → Report.

13.56.1 Learning objectives

- Fit a single decision tree, a random forest, and a gradient-boosted model on the same data, and compare their error.
- Read variable-importance output critically and know its biases.
- Explain when a tree ensemble is preferable to a linear model and when it is not.

13.56.2 Prerequisites

Cross-validation; regularisation.

13.56.3 Background

Tree-based models divide feature space by axis-aligned splits and fit a constant within each region. A single tree is interpretable but high variance; averaging many trees with resampling and random feature subsets gives random forests, which are among the strongest off-the-shelf methods on tabular data. Gradient boosting takes a different route: it fits trees sequentially, each correcting the residuals of the last, and usually wins on tabular benchmarks when carefully tuned.

Tree methods make few assumptions about the functional form of the response and handle interactions automatically. Their limitations are extrapolation (flat beyond the training range), variable-importance measures that can be misleading when features are correlated, and a general tendency to look better than they generalise unless honest resampling is used.

When reporting a random forest or gradient-boosted model, report the tuning procedure, the out-of-bag or CV error, the variable-importance method, and — if the goal is scientific — a permutation-based importance to cross-check the default Gini/gain importance.

13.56.4 Setup

```
library(tidyverse)
library(rpart)
library(ranger)
library(xgboost)
set.seed(42)
theme_set(theme_minimal(base_size = 12))
```

13.56.5 1. Goal

Predict `mpg` from `mtcars` with a single tree, a random forest, and gradient boosting; compare errors and importances.

13.56.6 2. Approach

```
d ← mtcars
ggplot(d, aes(wt, mpg, colour = factor(cyl))) +
  geom_point() + labs(colour = "cyl")
```

13.56.7 3. Execution

```
tree_fit ← rpart(mpg ~ ., data = d, cp = 0.01)
rf_fit   ← ranger(mpg ~ ., data = d, importance = "permutation",
                  num.trees = 500)
xg_fit   ← xgboost(
  data = as.matrix(d[, -1]), label = d$mpg,
  nrounds = 100, eta = 0.05, max_depth = 3,
  objective = "reg:squarederror", verbose = 0
)
```

13.56.8 4. Check

Honest error: CV for the tree and xgboost, OOB for ranger.

```
K ← 5
folds ← sample(rep(1:K, length.out = nrow(d)))
mse ← function(y, yhat) mean((y - yhat)^2)

get_err ← function() {
  err_tree ← err_rf ← err_xg ← numeric(K)
  for (k in 1:K) {
    tr ← folds ≠ k; te ← folds == k
    f_tr ← d[tr, ]; f_te ← d[te, ]
    t1 ← rpart(mpg ~ ., data = f_tr, cp = 0.01)
    r1 ← ranger(mpg ~ ., data = f_tr, num.trees = 500)
    x1 ← xgboost(
      data = as.matrix(f_tr[, -1]), label = f_tr$mpg,
      nrounds = 100, eta = 0.05, max_depth = 3,
      objective = "reg:squarederror", verbose = 0
    )
    err_tree[k] ← mse(f_te$mpg, predict(t1, f_te))
    err_rf[k]   ← mse(f_te$mpg, predict(r1, f_te)$predictions)
    err_xg[k]   ← mse(f_te$mpg, predict(x1, as.matrix(f_te[, -1])))
  }
  c(tree = mean(err_tree), rf = mean(err_rf), xgb = mean(err_xg))
}
errs ← get_err()
errs
```

```
tibble(var = names(rf_fit$variable.importance),
       imp = as.numeric(rf_fit$variable.importance)) ▷
  arrange(desc(imp)) ▷
  ggplot(aes(reorder(var, imp), imp)) + geom_col() +
  coord_flip() + labs(x = NULL, y = "permutation importance")
```

13.56.9 5. Report

> On `mtcars` (n = 32), 5-fold CV MSEs were `round(errs["tree"], 2)` (single tree), `round(errs["rf"], 2)` (random forest), and `round(errs["xgb"], 2)` (gradient boosting). Permutation importance from the random forest identified weight and displacement as the dominant predictors.

The CV errors are close because the sample is tiny and the problem is close to linear. On larger tabular problems the gap opens and gradient boosting typically wins after tuning; but the tiny-sample regime is where trees and boosting most often look like magic and almost always lie.

13.56.10 Common pitfalls

- Reporting the training error of a forest; always use OOB or CV.
- Trusting Gini/gain importance for correlated predictors.
- Letting xgboost overfit with low `eta` and high `nrounds` without early stopping.

13.56.11 Further reading

- Breiman L (2001), *Random forests*.
- Chen T, Guestrin C (2016), *XGBoost: A scalable tree boosting system*.

13.56.12 Session info

```
sessionInfo()
```

13.56.13 See also — chapter index

- Methods in this chapter
- Find your method (Chapter 0)