UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence

Department of mathematics informatics and geosciences

Introduction to Machine Learning

Lecturer:
Prof. Eric Medvet

Authors:

Christian Faccio
Andrea Spinelli

February 17, 2026

github.com/christianfaccio christianfaccio@outlook.it

Abstract

As a student of the “Data Science and Artificial Intelligence” master’s degree at the University of Trieste, I have created these notes to study the course “Introduction to Machine Learning” held by Prof. Eric Medvet. The course aims to provide an introduction to the field of machine learning, covering the fundamental concepts in theoretical terms. The topics are the followig:

- Supervised learning techniques
- Unsupervised learning techniques
- Dimensionality reduction (not covered in the lectures)
- Introduction to NLP

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in this field.

Contents

1 Basic Concepts

1.1 What is Machine Learning?

Machine learning is a research area at the intersection of computer science, statistics and optimization; its goal is to design systems that make useful decisions from data.

Definition: ***Machine Learning***

Machine Learning is the science of getting computers to learn without being explicitly programmed.

Each part of this sentence hides important questions: what kind of *science* is this field? who is actually building and running these computer systems? what does it mean for a computer to *learn*, and learn *what* exactly? in which sense is the system not being explicitly programmed?

To answer these questions we start from a very simple view of learning as *decision making* based on observations. Let's formalize the decision making process:

$$y = f(x)$$

- x is the entity about which the decision has to be made
- y is the decision
- f is the procedure that, given an x results in a decision y

We call x an **observation** and y the **response**. When we view f as an information-processing system, x is often referred to as the **input** and y as the **output**. The observation x may also be called a **data point** or an **instance**.

The central point is *what* the computer learns: how to make a decision y about an observation x, i.e. a function $f : X \to Y$. We want the computer not only to learn f but to use it; in practice, once f is fixed, we use it to predict y for new x. The function is therefore often denoted f_{predict}. When we use it on new data, we are in the **prediction phase**; the value it outputs is a prediction for the (possibly unknown) true response, often written $\hat{y}$.

Definition: ***Machine Learning (refined)***

Machine Learning is the science of getting computers to learn $f_{\text{predict}} : X \to Y$ **autonomously**.

Here, "autonomously" means that f_{predict} is *not* written by a human: we ask the computer to *find* a program (a function f) that behaves as desired, instead of writing it by hand. Formally:

1. consider the set of all functions from X to Y, $\mathcal{F}_{X \to Y} = \{f : X \to Y\}$;
2. choose one $f \in \mathcal{F}_{X \to Y}$ that does what is expected.

Step 2 is what we mean by learning; specifying "what is expected" in general is difficult, and in practice we need some form of **supervision** to guide the search for a good f.

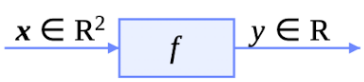


Figure 1.1: Abstract view of the learning process.

1.1.1 Learning paradigms and supervision

The way supervision is provided leads to different learning paradigms.

- **Supervised learning**: we observe example pairs (x, y) of observations and desired responses, and we want f to reproduce this behaviour.
- **Unsupervised learning**: we only observe x's and look for structure or regularities in the data without explicit labels.
- **Reinforcement learning**: the system interacts with an environment and receives a **reward** signal instead of explicit target responses.

Supervised learning

We focus on the supervised case. Examples are pairs (x, y); the finite set of examples used for learning is the **training set** (or **learning set**), often denoted D_{learn}. A training set is a particular kind of **dataset**, i.e. a collection of observations and responses. It should be compatible with the chosen X and Y; more precisely, each pair in D_{learn} must belong to $X \times Y$. We write:

$$f_{\text{predict}} = f_{\text{learn}}(D_{\text{learn}})$$

to highlight that the predictor is produced by applying a learning procedure f_{learn} to the data. Conceptually we distinguish two phases:

- the **learning phase**, in which f_{learn} processes D_{learn} and outputs f_{predict};
- the **prediction phase**, in which f_{predict} is used to predict responses for new observations.

Definition: ***Supervised Learning Technique***

A ***Supervised Learning Technique*** is a method that, given a learning set $D_{\text{learn}} \in \mathcal{P}^*(X \times Y)$, returns a predictor $f_{\text{predict}} \in \mathcal{F}_{X \to Y}$.

Where $X \times Y$ is the cartesian product of X and Y, and $\mathcal{P}(A)$ is the power set of A (the set of all subsets of A), and $\mathcal{P}^*(A)$ is the power set of A with duplicates.

Different supervised learning techniques can be compared along several axes:

- **applicability**, i.e. the kinds of X and Y they can handle (e.g. $X = \mathbb{R}^p$ or Y real-valued);
- **efficiency**, i.e. the computational resources they require as a function of $|D_{\text{learn}}|$;
- **effectiveness**, i.e. the quality of the predictor they learn.
- **interpretability**, i.e. the ease with which the predictor can be understood by humans.

Supervised learning as optimisation

A particularly fruitful viewpoint is to see supervised learning as an optimisation problem. A supervised learning technique $f_{\text{learn}} : \mathcal{P}^*(X \times Y) \to \mathcal{F}_{X \to Y}$ can be seen as a form of optimisation. The general approach consists of two steps:

1. consider $\mathcal{F}_{X \to Y} = \{f, f : X \to Y\}$;
2. find the one $f_{\text{predict}} \in \mathcal{F}_{X \to Y}$ that **works best** on D_{learn}.

Could we use a general optimisation technique? In principle yes, but there are practical obstacles. The space X (and maybe Y) might be infinite (for example $X = \mathbb{R}^p$); the space $X \times Y$ is "more" infinite; and the space $\mathcal{F}_{X \to Y}$ of all functions from X to Y is enormously larger still.

Templating f and learning a model

A practical way to make the search feasible is to restrict $\mathcal{F}_{X \to Y}$ to a smaller family of functions of a particular structure. We introduce a **template** function f' and a reduced hypothesis space $\mathcal{F}'_{X \to Y} \subseteq \mathcal{F}_{X \to Y}$ consisting of all functions that are obtained by instantiating this template.

Most parts of f' are fully specified; only some components are left *undefined* and will later be chosen from data. The template f' can be used for prediction only after these undefined parts have been fixed.

Example: *Univariate linear regression as a template*

Suppose $X = Y = \mathbb{R}$ and consider the family of affine functions

$$f'(x;a,b) = ax + b, \qquad a,b \in \mathbb{R}.$$

Here f' is the template; the parameters a and b are the undefined parts. This is called *univariate linear regression*: it is *univariate* because X has one dimension, a regression problem because $Y = \mathbb{R}$, and *linear* because of the affine template.

We collect the free parameters of the template in a vector $m \in M$; this vector is called the **model**. Once m is fixed, the template becomes a concrete predictor

$$f_{\text{predict}}(x) = f'_{\text{predict}}(x,m),$$

so that m completely determines how y depends on x.

In the most general (non-templated) view a supervised learning technique is a function

$$f_{\text{learn}} \colon \mathcal{P}^*(X \times Y) \to \mathcal{F}_{X \to Y}, \qquad f_{\text{predict}} \colon X \to Y,$$

that maps a learning set D_{learn} to a predictor f_{predict}.

For techniques based on a template we separate these two roles:

$$f'_{\text{learn}} \colon \mathcal{P}^*(X \times Y) \to M, \qquad f'_{\text{predict}} \colon X \times M \to Y.$$

The learning technique is now defined by the pair $(f'_{\text{learn}}, f'_{\text{predict}})$: from data D_{learn}, the procedure f'_{learn} selects a model $m \in M$, and the templated predictor $f'_{\text{predict}}(\cdot, m)$ can then be used to map any new observation x to a response y.

Figure 1.2: the learning phase produces a model m, which is used by f'_{predict} to compute y from x.

1.1.2 Notation and terminology

Several terms are used interchangeably in literature to describe the process of obtaining a model from data. We say that a model m is **learned**, **trained**, or **fitted** on a dataset D; all three expressions mean that m is adjusted until it works well on the examples in D. Formally, a model is one specific $m \in M$ that has been found through the learning process. However, the word "model" is sometimes used more loosely to denote a also an untrained template.

Supervised learning techniques can be categorized according to the nature of X, Y, and M. The most important distinction concerns the output space Y:

- **Classification**: Y is a finite set with no ordering. The label y is called **categorical** (or **nominal**). If $|Y| = 2$ we speak of **binary classification**; otherwise it is **multiclass classification**.
- **Regression**: $Y = \mathbb{R}$ (or $Y \subseteq \mathbb{R}$). The response y is called a **numerical variable**.

Regarding the input space X, common cases include:

- **Multivariate**: $X = X_1 \times X_2 \times \cdots \times X_p$, where each X_j is either $\mathbb{R}$ or a finite unordered set. Each observation x is a p–tuple, and each component x_j is either numerical or categorical.
- **Text mining**: X is the set of all strings (or sequences of symbols).

Variables terminology

In the multivariate case $X = X_1 \times X_2 \times \cdots \times X_p$:

- each component x_j is called an **independent variable**. Alternative names are **feature** (since it is a feature of an observation $x \in X$), **attribute**, or **predictor** (since it helps predict y).
- y is called the **dependent variable** or **response variable**, since it is hoped to depend on x.

$$D = \begin{pmatrix} x_1^{(1)} & \dots & x_j^{(1)} & \dots & x_p^{(1)} & y^{(1)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_1^{(i)} & \dots & x_j^{(i)} & \dots & x_p^{(i)} & y^{(i)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_1^{(n)} & \dots & x_j^{(n)} & \dots & x_p^{(n)} & y^{(n)} \end{pmatrix}$$

- $x^{(i)}$ is the i-th **observation**
- $\left\{ x_j^{(i)} \right\}_i$ are the values of the j-th **feature**
- $x_j^{(i)}$ is the **value** of the j-th feature for the i-th observation (recall, order does not matter in D)
- $y^{(i)}$ is the **response** for the i-th observation
  - if y is categorical $\rightarrow$ **class label**

Size of the "problem"

The common notation for the size of a multivariate dataset (i.e. $X = X_1 \times X_2 \times \cdots \times X_p$) is:

- n: number of observations
- p: number of (independent) variables

On the assumption that a dataset D implicitly defines the problem (since it bounds X and Y and hence $\mathcal{F}_{X \to Y}$), n and p also describe the size of the problem.

Common models we will see

Throughout this course, we will encounter several families of learning techniques:

- **Tree-based methods**
  Tree-based methods work with multivariate inputs $X = X_1 \times \cdots \times X_p$, where each component can be categorical or numerical. They handle both classification (binary or multiclass) and regression tasks.
- **Support Vector Machines (SVM)**
  Support Vector Machines (SVM) are designed for numerical inputs $X = \mathbb{R}^p$ and are primarily used for binary classification.
- **k-Nearest Neighbours (kNN)**
  k-Nearest Neighbours (kNN) can be applied to any input space X equipped with a similarity metric (including $X = \mathbb{R}^p$). It supports both classification and regression.
- **Naive Bayes**
  Naive Bayes is suited for multivariate inputs $X = X_1 \times \cdots \times X_p$ where each component is categorical (though hybrid cases exist). It is used for classification tasks.

If none of these techniques fits the problem we will also cover several adaptation methods:

- encoding categorical variables so that techniques designed for $X = \mathbb{R}^p$ can be applied to mixed-type inputs;
- transforming text data into numerical representations, enabling techniques for $X = \mathbb{R}^p$ to handle string inputs;
- extending binary classifiers ($|Y| = 2$) to multiclass problems ($|Y| \geq 2$).

1.1.3 Machine learning systems

In realistic applications, a learning algorithm is only one component of a larger information-processing pipeline. Raw data must often be **pre-processed** before learning (for example by normalisation, feature extraction or handling missing values), and the outputs of the predictor may need to be **post-processed** before they are used to take actions.

Definition: ***ML system***

An ***ML system*** is an information-processing system that combines a (supervised) learning technique with suitable pre-processing and post-processing steps.

The designer of the ML system must choose the learning technique, the pre/post-processing steps, and the relevant parameters. The typical workflow consists of the following steps:

1. **Decide**: determine whether the problem at hand actually calls for an ML approach;
2. **Choose the learning paradigm**: supervised, unsupervised, reinforcement;
3. **Define the problem** (problem statement):
   - define the input space X and output space Y;
   - define a *way for assessing* solutions. This should be done before designing, so that it is applicable to any compatible ML solution;
4. **Design** the ML system:
   - choose a learning technique;
   - choose or design pre- and post-processing steps;
5. **Implement** the ML system:
   - execute the learning and prediction phases;
   - obtain the data;
6. **Assess** the ML system on relevant tasks.

Steps 4-6 are usually iterated many times during development.

When to use Machine Learning?

Machine learning is not always the right tool for every problem. It is worth considering both *why* we might want a machine to execute f_{predict}, and *why* we might want the machine to learn it autonomously rather than having a human design it. There are several scenarios where running f_{predict} on a machine is preferable:

- The response y must be computed **very quickly**, faster than a human could manage.
- The prediction must be made in a **dangerous/inaccessible place** where humans can't be present.
- The **value of** y **is very low**, making human involvement economically unjustifiable.
- There is concern that a human would be **biased** in deciding y.

Even when f_{predict} is executed by a machine, it could still be designed by a human. The question then becomes: why should the machine *learn* the predictor instead?

- The relationship between x and y may be too complex or poorly understood.
- A human-designed predictor would be **too costly or slow** to develop.
- A human-designed predictor is simply **not good enough**.

When deciding whether to adopt ML, three factors should be weighed:

- **Efficiency**: how quickly and cheaply can predictions be made?
- **Effectiveness**: how accurate are the predictions?
- **Human dignity**: are there ethical or social reasons to prefer human involvement?

2 Supervised Learning

2.1 Assessment

When assessing something with respect to a given **goal**, three main axes are considered:

- **Effectiveness**: to which degree is the goal achieved?
- **Efficiency**: how many resources are consumed for achieving the goal (to some degree)?
- **Interpretability** (or explainability): to which degree is the way the goal is achieved explainable?

Given an axis a of assessment, there are two purposes:

- **Absolute assessment**: does something meet expectations in terms of a? (is effective *enough*?)
- **Comparison**: is a thing better than another in terms of a? (is m_1 more effective than m_2?)

If the outcome of assessment is a **quantity** (with monotonic semantics), then comparison reduces to checking $>$ or $<$, while absolute assessment requires establishing a threshold and then checking against it. For this reason, we want assessment to produce a number.

A **ML system** can be seen as a composite learning technique with two running modes: **training** in which it tunes itself, and **testing** in which it makes decisions. Its goals are to tune properly (so that it makes good decisions after tuning) and to make good decisions.

A **supervised learning technique** (SLT) is a pair $f_{\text{learn}}, f_{\text{predict}}$; its goals are to learn a good f_{predict} (one that makes good decisions) and to make good decisions. A **model**, instead, has the single goal of making good decisions when used in an f'_{predict}.

Ultimately, **effectiveness is about making good decisions**. To measure the effectiveness of a model and to have a number as output, we need to compare our system with the real system that generated the data (assuming there is one).

Model vs Real System

How do we measure whether f'_{predict} is making good decisions? Recall that f_{predict} (possibly implemented via f'_{predict} and a model m) attempts to capture the dependency of y on x.

The key underlying assumption is that **y depends on x**. That is, there exists a real system $s : X \to Y$ which, given an input x, produces an output y; formally, $s \in \mathcal{F}_{X \to Y}$. Alternatively, there may exist a system $s^{-1} : Y \to X$ that, given a y, produces an x based on it.

Figure 2.1: A templated $f'_{\text{predict}} : X \times M \to Y$ with a fixed model m is an $f_{\text{predict}} : X \to Y$.

To compare m and s, we adopt a **comparison of behaviors** approach:

1. Collect some examples of the behavior of s
2. Feed m with the same examples
3. Compare responses of s and m

Ideally, we want the comparison (3) to produce a number. This procedure can be formalized as:

$$f_{\text{comp-behavior}} : \mathcal{F}_{X \to Y} \times \mathcal{F}_{X \to Y} \to \mathbb{R}$$

or, to highlight the presence of a model in a templated f'_{predict}:

$$f_{\text{comp-behavior}} : \mathcal{F}_{X \times M \to Y} \times M \times \mathcal{F}_{X \to Y} \to \mathbb{R}$$

In both cases:

```
function comp-behavior(f_predict, s) {
    {x^(i)}_i ← collect()
    {y^(i)}_i ← foreach({x^(i)}_i, s)
    {ŷ^(i)}_i ← foreach({x^(i)}_i, f_predict)
    v_eff ← comp-resp({{y^(i), ŷ^(i)}}_i)
    return v_eff
}
```

More correctly:
$\{(y^{(i)}, \hat{y}^{(i)})\}_i = \forall(\{x^{(i)}\}_i, \text{both}(\cdot, s, f_{\text{predict}}))$ with $f_{\text{both}} : X \times \mathcal{F}^2_{X \to Y} \to Y$ and $f_{\text{both}}(x, f_1, f_2) = (f_1(x), f_2(x))$.
Note that $f_{\text{comp-behavior}}$ is a partially abstract function: f_{collect} and $f_{\text{comp-resp}}$ are **abstract** (i.e., not concretely specified). We reason about the effectiveness and efficiency of $f_{\text{comp-behavior}}$ based on concrete choices for f_{collect} and $f_{\text{comp-resp}}$:

- **Effectiveness**: to which degree $f_{\text{comp-behavior}}$ measures if m behaves like s
- **Efficiency**: how many resources are consumed to apply $f_{\text{comp-behavior}}$

Warning: *Remarks on $f_{collect}$*

The data collection step is crucial:

- small $n \to$ **poor effectiveness, high efficiency**
- large $n \to$ **good effectiveness, poor efficiency**

Also, **coverage** (how data represents the real system) affects effectiveness regardless of n.

Formally:

$$f_{\text{comp-resps}} : \mathcal{P}^*(Y^2) \to \mathbb{R}$$

where $\{(y^{(i)}, \hat{y}^{(i)})\}_i \in \mathcal{P}^*(Y^2)$ is a multiset of pairs of y.
Depends only on Y, not on X!

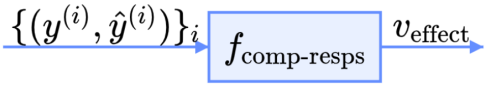


Figure 2.2: Comparing the responses

Example: *Performance Indexes*

Classification:

- all (i.e., agnostic with respect to $|Y|$): **error**, **accuracy**
- binary: **FPR**, **FNR** (and variants), EER, AUC
- multiclass: **weighted accuracy**

Regression:

- **MAE**, **MSE**, MRE, R2

2.1.1 Classification

In Classification, Y is a finite set with no ordering.
The **Classification Error** measures the fraction of incorrect predictions:

$$f_{err}\left(\left\{(y^{(i)},\hat{y}^{(i)})\right\}_{i=1}^{n}\right)=\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}(y^{(i)}\neq\hat{y}^{(i)})$$

where $\mathbf{1}:\{\text{false},\text{true}\}\to\{0,1\}$ is the **indicator function**:

$$\mathbf{1}(b)=\begin{cases}1 & \text{if } b \text{ is true}\\ 0 & \text{otherwise}\end{cases}$$

The lower the error, the better.

The **Classification Accuracy** measures the fraction of correct predictions:

$$f_{acc}\left(\left\{(y^{(i)},\hat{y}^{(i)})\right\}_{i=1}^{n}\right)=\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}(y^{(i)}=\hat{y}^{(i)})=1-f_{err}\left(\left\{(y^{(i)},\hat{y}^{(i)})\right\}_{i}\right)$$

The higher the accuracy, the better.

Extreme cases for accuracy (and error) are:
- m is s: **perfect model** $\to f_{acc}=1$, $f_{err}=0$
- m is random: does not model any dependence $\to f_{acc}\approx\frac{1}{|Y|}$

Example: ***Classifier Bounds***

Since accuracy lies in $[0,1]$, we can establish reference points for evaluating classifiers. Recall that f_{acc} measures how well a model m approximates the true system s. The **ideal extreme cases** in practice are: (i) m is random and captures no dependency of y on x (lower bound), or (ii) m perfectly models s (upper bound).

- **Lower Bounds:**
  - ***Random Classifier:*** pick uniformly at random $\quad \text{acc}\sim\frac{1}{|Y|}$

$$f_{random}(x)=y_i,\ \ i\sim U(\{1,\ldots,|Y|\})$$

  - ***Dummy Classifier:*** pick the most frequent class $\quad \text{acc}\sim\max_{y\in Y}Fr(y,\{y^{(i)}\}_i)$

$$f_{dummy}(x)=\arg\max_{y}\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}(y=y^{(i)})=\arg\max_{y\in Y}Fr(y,\{y^{(i)}\}_i)$$

- **Upper Bounds:**
  - ***Perfect Classifier:*** deterministic ideal $\quad \text{acc}=1$

$$f_{perfect}(x)=s(x)$$

  - ***Bayes Classifier:*** stochastic ideal $\quad \text{acc}=\max_{y\in Y}\Pr(s(x)=y\mid x)$

$$f_{Bayes}(x)=\arg\max_{y}\Pr(s(x)=y\mid x)$$

Note: Since the world is not deterministic, the Bayes classifier (stochastic) may be a better upper bound than the perfect classifier.

2.1.2 Binary Classification

In binary classification, we assign standard names to the two possible y values: **positive** (denoted pos) and **negative** (denoted neg). The common practice is to associate positive with the **rarest case**; otherwise, if no rarest case exists, one should clearly state which class is considered positive. The goal of FPR and FNR is to measure the error on each of the two classes separately.

Definition: *FPR and FNR*

The **False Positive Rate (FPR)** is the rate of negatives wrongly classified as positives:

$$\mathrm{FPR} = \frac{\sum_i \mathbf{1}(y^{(i)} = \mathrm{neg} \wedge \hat{y}^{(i)} \neq y^{(i)})}{\sum_i \mathbf{1}(y^{(i)} = \mathrm{neg})} = \frac{\mathrm{FP}}{\mathrm{N}}$$

The **False Negative Rate (FNR)** is the rate of positives wrongly classified as negatives:

$$\mathrm{FNR} = \frac{\sum_i \mathbf{1}(y^{(i)} = \mathrm{pos} \wedge \hat{y}^{(i)} \neq y^{(i)})}{\sum_i \mathbf{1}(y^{(i)} = \mathrm{pos})} = \frac{\mathrm{FN}}{\mathrm{P}}$$

For both: the codomain is $[0,1]$, and **the lower, the better**.

In the shorthand notation: FP and FN are the counts of false positives and false negatives (requiring both $y^{(i)}$ and $\hat{y}^{(i)}$), while P and N are the counts of actual positives and negatives (requiring $y^{(i)}$).

		Test Result	
		0	1
Ground Truth	0	TN (True Negative)	FP (False Positive)
	1	FN (False Negative)	TP (True Positive)

Figure 2.3: Confusion matrix for binary classification.

The **True Positive Rate (TPR)** and **True Negative Rate (TNR)** measure correct classifications:

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = 1 - \mathrm{FNR} \qquad \mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = 1 - \mathrm{FPR}$$

For both, **the greater, the better** (codomain is $[0,1]$).

Error and accuracy can be expressed in terms of these rates:

$$\mathrm{Err} = \frac{\mathrm{FP} + \mathrm{FN}}{\mathrm{P} + \mathrm{N}} = \frac{\mathrm{P} \cdot \mathrm{FNR} + \mathrm{N} \cdot \mathrm{FPR}}{\mathrm{P} + \mathrm{N}} \qquad \mathrm{Acc} = 1 - \mathrm{Err} = \frac{\mathrm{P} \cdot \mathrm{TPR} + \mathrm{N} \cdot \mathrm{TNR}}{\mathrm{P} + \mathrm{N}}$$

Warning: *Accuracy alone is dangerous*

Consider a rare disease affecting ≈ 2 persons per 1000. A trivial test that *always* predicts "no disease" achieves Acc $= 99.8\%$, FPR $= 0\%$, but FNR $= 100\%$. This illustrates why **accuracy alone in binary classification can be misleading**: always report FPR and FNR (or related metrics) alongside accuracy.

Definition: *Balanced Data*

A dataset is **balanced** with respect to y if the frequency of each class is roughly the same. In skewed (unbalanced) datasets, accuracy becomes particularly unreliable.

Other commonly used metrics include:

- **Precision**: $\frac{\text{TP}}{\text{TP}+\text{FP}}$ (fraction of predicted positives that are correct)
- **Recall** (Sensitivity, TPR): $\frac{\text{TP}}{\text{TP}+\text{FN}} = \text{TPR}$
- **Specificity** (TNR): $\frac{\text{TN}}{\text{TN}+\text{FP}} = \text{TNR}$
- **F1 Score**: $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$ (harmonic mean)

In hypothesis testing terminology: FPR is the **Type I error** and FNR is the **Type II error**.

Warning: *Cost of the error*

Once $f_{predict}$ outputs a y, some action is taken, and if the action is wrong, there is some cost to be paid wrt the correct action.

$$c = c_{FP} \times FPR \times N + c_{FN} \times FNR \times P$$

If one knows the costs and N,P, it is possible to compute the overall cost and find a good trade-off for FPR and FNR.

Balancing FPR and FNR

In many binary classification problems we want to *balance* the two kinds of mistakes (FP and FN) according to their relative importance. For many learning techniques, the model internally computes *more* than just the final class label: for classification it often computes a *probability* for each class, and then converts it into a label by comparing it with a *threshold*.

Probabilistic classifiers, instead of returning only a label $f'_{\text{predict}} : X \times M \to Y$, return a *discrete probability distribution* over Y:

$$f''_{\text{predict}} : X \times M \to P_Y, \qquad \text{output: } p = f''_{\text{predict}}(x, m)$$

where P_Y is the set of all discrete distributions over Y.

The corresponding hard prediction is obtained by picking the most probable class:

$$f'_{\text{predict}}(x, m) = \arg\max_{y \in Y} p(y) = \arg\max_{y \in Y} f''_{\text{predict}}(x, m)(y).$$

In the **binary** case $Y = \{\text{pos}, \text{neg}\}$, the distribution p is completely determined by the probability of the positive class, $\hat{p}_{\text{pos}} = p(\text{pos}) \in [0, 1]$, since $p(\text{neg}) = 1 - \hat{p}_{\text{pos}}$. We can define a dedicated predictor:

$$f'''_{\text{predict}} : X \times M \to [0, 1], \qquad \hat{p}_{\text{pos}} = f'''_{\text{predict}}(x, m),$$

and then use a threshold τ to obtain a label:

$$f_{\text{predict}}(x, m) = \begin{cases} \text{pos} & \text{if } \hat{p}_{\text{pos}} \geq \tau, \\ \text{neg} & \text{otherwise.} \end{cases}$$

Most software libraries use the ***default threshold*** $\tau = 0.5$, which corresponds to choosing the most probable class. However, *there is nothing special about* 0.5; other values of τ may be better when the costs c_{FP}, c_{FN} or the class frequencies are unbalanced.

In the binary case, the closer $\hat{p}_{\text{pos}}$ is to 0.5, the less confident the model is in its decision; values close to 0 or 1 indicate high confidence. We can define a simple **confidence** measure

$$\text{conf}(x,m) = \frac{|\hat{p}_{\text{pos}} - 0.5|}{0.5} \in [0,1],$$

so that conf $= 0$ means total uncertainty (the model is "on the fence") and conf $= 1$ means maximal confidence. This is a basic form of *local explainability* for the prediction on a single x.

If we **change the threshold** τ, we change the trade-off between FPR and FNR for the *same* model m and dataset $\{(x^{(i)}, y^{(i)})\}_i$.

- If we **increase** τ (for example from 0.5 to 0.7), the classifier predicts "pos" less often:
  - many borderline pos become neg; $\rightarrow$ FNR tends to **increase**, FPR tends to **decrease**.
- If we **decrease** τ (for example from 0.5 to 0.3), the classifier predicts "pos" more often:
  - many borderline neg become pos; $\rightarrow$ FNR tends to **decrease**, FPR tends to **increase**.

Intuitively, a high threshold is conservative: the classifier says "positive" only when very sure; a low threshold is aggressive: it prefers to avoid false negatives, at the cost of more false positives.

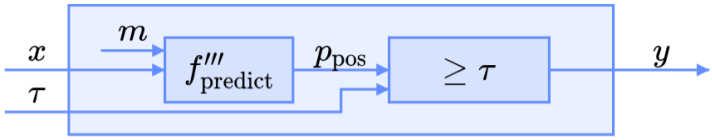


Figure 2.4: Effect of the decision threshold on the decision boundary.

The threshold therefore allows us to *navigate the trade-off* between FPR and FNR, and possibly to account for the error cost

$$c(\tau) = c_{FP} \cdot FPR(\tau) \cdot N + c_{FN} \cdot FNR(\tau) \cdot P.$$

In principle, if c_{FP}, c_{FN}, P, N are known, one could choose τ so as to minimise $c(\tau)$.

Equal Error Rate (EER)

Often one is interested in a single operating point that balances the two kinds of errors.

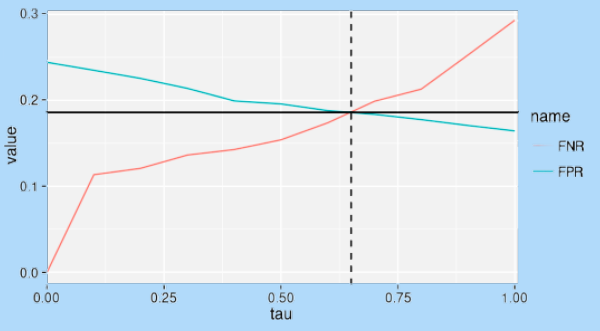
Definition: ***Equal Error Rate***

For a model m and a dataset $\{(x^{(i)}, y^{(i)})\}_i$, the ***Equal Error Rate (EER)*** is the value

$$\text{EER} = FPR(\tau_{\text{EER}}) = FNR(\tau_{\text{EER}})$$

where τ_{EER} is the threshold for which

$$FPR(\tau) = FNR(\tau)$$



The EER is an *index* of classifier quality: **the lower, the better** (like error). Its codomain is $[0,1]$, and in practice values lie in $[0, 0.5]$.

The EER is particularly useful when the two types of errors have comparable costs.

Receiver Operating Characteristic (ROC) curve

The behaviour of a binary classifier as the threshold τ varies can be summarised by the *ROC curve*.

Definition: ***ROC curve (Receiver Operating Characteristic)***

Given a fitted model m and a dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$, the ***ROC curve*** is the plot of

$$\big(FPR(\tau), TPR(\tau)\big)$$

for all possible values of the threshold τ.

Each threshold τ gives one point on the ROC plane; moving τ from 0 to 1 traces a curve. The diagonal line $TPR = FPR$ corresponds to a *random classifier*. A good classifier has a ROC curve that lies well above this diagonal and is close to the top-left corner $(FPR = 0, TPR = 1)$.

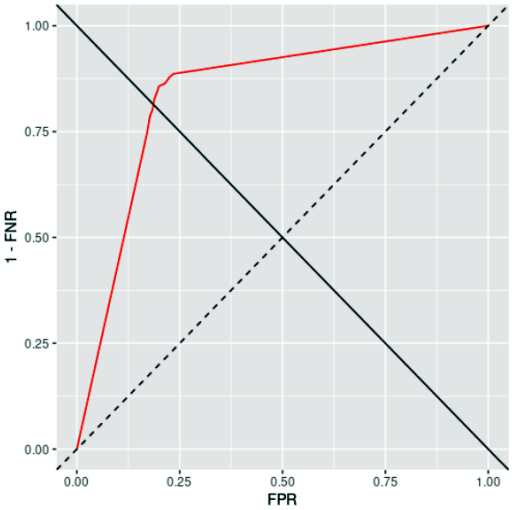


Figure 2.5: ROC curve

- ***Red line***: ROC curve; each point corresponds to (FPR, TPR) for a given τ.
- ***Solid black line***: $FPR = FNR$; its intersection with the red line gives the EER, and the top-left point $(FPR = FNR = 0)$ is the perfect classifier.
- ***Dashed black line***: $TPR = FPR$; the intersection with the solid line is the random classifier, points along it represent random classifiers with $\tau \neq 0.5$.
- A healthy classifier's ROC should ***never lie to the right*** of this diagonal.

Area Under the Curve (AUC)

Sometimes the threshold τ at which the classifier will be used in practice is not known in advance, or may vary. In such cases it is useful to have a *threshold-independent* measure of performance.

Definition: ***Area Under the ROC Curve (AUC)***

The ***AUC*** is the area under the ROC curve:

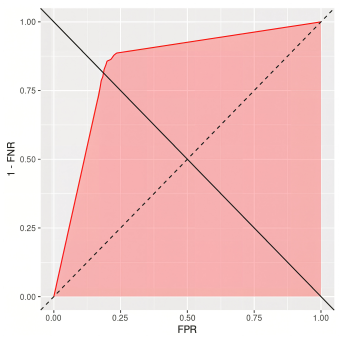
$$\text{AUC} = \int_0^1 TPR(FPR^{-1}(u))\,\mathrm{d}u,$$

computed numerically from the ROC points.

The AUC provides a single scalar summary of classifier performance across all thresholds.

It satisfies:

- AUC $= 0.5$ for a random classifier (the diagonal baseline);
- AUC $= 1$ for a perfect classifier (ideal separation);
- in practice, $0.5 \leq$ AUC ≤ 1 for any reasonable model;
- ***the larger, the better***: higher AUC indicates better discrimination.



Choosing the threshold values

To compute EER and AUC, and thus evaluate FPR and TPR across many values of τ, we need:

- a fitted model m and the prediction function f_{predict};
- a labelled dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$;
- a sequence of thresholds $(\tau_j)_{j=0}^{k}$.

Three common strategies to choose the threshold sequence are:

1. **Uniform grid on** $[0,1]$: choose $k+1$ evenly spaced points,

$$\tau_j = \frac{j}{k}, \quad j = 0, 1, \ldots, k.$$

2. **Uniform grid on the prediction range**: let $\hat{p}^{(i)} = f_{\text{predict}}(x^{(i)}, m)$ and define

$$\tau_{\min} = \min_i \hat{p}^{(i)}, \qquad \tau_{\max} = \max_i \hat{p}^{(i)}.$$

   Then take $k+1$ evenly spaced points in $[\tau_{\min}, \tau_{\max}]$:

$$\tau_j = \tau_{\min} + \frac{j}{k}(\tau_{\max} - \tau_{\min}), \quad j = 0, 1, \ldots, k.$$

3. **Unique prediction values**: sort the predicted probabilities $\{\hat{p}^{(i)}\}$ and use each distinct value (or their midpoints) as a threshold.

The third approach is the most efficient: it yields exactly the distinct operating points that the classifier can achieve on the dataset, without redundant evaluations.

Warning: ***Which index should I use?***

- If the operating threshold τ is fixed (for example $\tau = 0.5$) and the error costs are known, evaluate FPR, FNR (and possibly the cost $c(\tau)$) at that threshold.
- If the threshold will be tuned later or may change over time, AUC is often a more informative index than accuracy.
- If you want a *single* threshold-independent scalar index, EER and AUC are common choices; AUC is more widely used.

Confusion matrix

Given a multiset $\{(y^{(i)}, \hat{y}^{(i)})\}_{i=1}^{n}$ of true and predicted labels, the ***confusion matrix*** is a $|Y| \times |Y|$ table where:

- each row corresponds to a true class $y \in Y$;
- each column corresponds to a predicted class $\hat{y} \in Y$;
- the entry $c_{y,\hat{y}}$ counts how many examples have true label y and predicted label $\hat{y}$.

$Y = \{\text{pos}, \text{neg}\}$

$y^{(i)}$	$p^{(i)}_{\text{pos}}$	$\hat{y}^{(i)}$	out
●	0.49	●	FN
●	0.29	●	TN
●	0.63	●	TP
●	0.51	●	TP
●	0.52	●	TP
●	0.47	●	TN
●	0.94	●	TP
●	0.75	●	TP
●	0.53	●	FP
●	0.45	●	TN

For this case:

$y \backslash \hat{y}$	●	●
●	5	1
●	1	3

For binary classification:

$y \backslash \hat{y}$	pos	neg
pos	TP	FN
neg	FP	TN

In general it holds, being c the confusion matrix:

- the **accuracy** is the ratio between the sum of the diagonal and the sum of the matrix:

$$\text{Acc} = \frac{\|\text{diag}(c)\|_1}{\|c\|_1};$$

- **TPR** is the ratio of $c_{\text{pos,pos}}$ on the sum of the first row, i.e., the row for which $y = \text{pos}$;
- **TNR** is the ratio of $c_{\text{neg,neg}}$ on the sum of the second row, i.e., the row for which $y = \text{neg}$.

2.1.3 Multiclass Classification and Regression

In multiclass classification ($|Y| > 2$) accuracy and error can still be defined as in Section 2.1.2, but they may hide class imbalances. A useful alternative is the *weighted* (or *balanced*) accuracy.

Weighted accuracy for multiclass classification

Let Acc_y be the accuracy computed *on class y only*, that is, the fraction of examples of class y which are correctly classified. The ***weighted accuracy*** (here with uniform weights) is

$$f_{acc} = \frac{1}{|Y|} \sum_{y \in Y} Acc_y.$$

In terms of confusion matrix, Acc_y is the ratio between the element on the diagonal for class y and the sum of the row for class y. The weighted accuracy gives each class the same importance, regardless of its frequency in the dataset; this is particularly useful when the dataset is unbalanced.

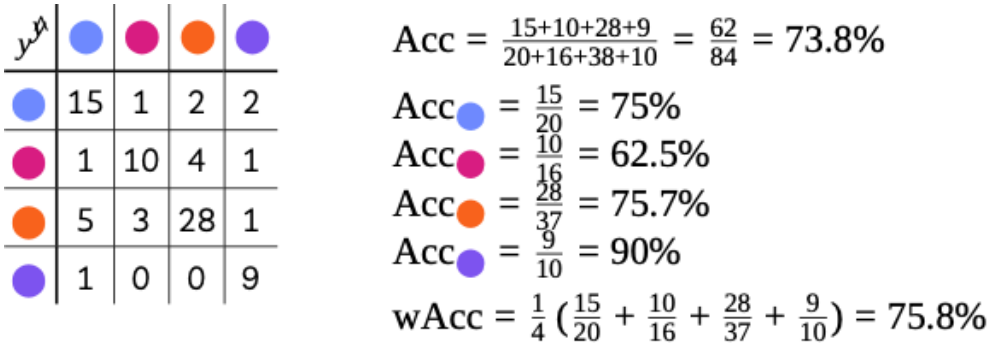


Figure 2.6: Multiclass confusion matrix. Rows correspond to true classes; cols to predicted ones.

Errors in regression

In regression the output space is numerical; a prediction can be *more or less wrong* depending on how far it is from the true value. Several error measures are commonly used.

Let $\{(y^{(i)}, \hat{y}^{(i)})\}_{i=1}^{n}$ be the multiset of true and predicted responses. Then:

- **Mean Absolute Error (MAE)** $$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y^{(i)} - \hat{y}^{(i)}|$$
- **Mean Squared Error (MSE)** $$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$
- **Root Mean Squared Error (RMSE)** $$\text{RMSE} = \sqrt{\text{MSE}}$$
- **Mean Absolute Percentage Error (MAPE)** $$\text{MAPE} = \frac{1}{n} \sum_{i=1}^{n} \frac{|y^{(i)} - \hat{y}^{(i)}|}{|y^{(i)}|}$$

For all these indexes:

- the *lower*, the better;
- the domain is $[0, +\infty)$ (MAPE might be ∞);
- MAE and RMSE **retain the unit of measure** of y (e.g., if y is in meters, MAE is in meters);
- MAPE is **scale-independent** (dimensionless);
- MSE and RMSE are **more influenced** by observations with **large errors**;
- MAPE "does not work" if the true $y^{(i)} = 0$.

2.1.4 Assessing Learning Techniques

A supervised learning technique is effective if it learns a model that *generalises*, i.e. that performs well on new (unseen) observations drawn from the same real system.

- an **effective** learning technique is a pair $(f'_{\text{learn}}, f'_{\text{predict}})$ that learns a good model m from data;
- a good **model** is one whose behaviour is close to the behaviour of the real system s.

To measure the effectiveness of a learning technique, one must:

- learn a model m from a dataset (learning phase);
- evaluate the model on data (prediction phase plus performance index).

The core design choice concerns the split of the available dataset D into training and testing portions.

Definition: *Effectiveness of a learning technique on a dataset*

Let $(f'_{\text{learn}}, f'_{\text{predict}})$ be a learning technique and let D be a dataset. An assessment procedure returns a number

$$\text{Eff}\left(f'_{\text{learn}}, f'_{\text{predict}}; D\right) \in \mathbb{R}$$

by (i) learning a model on (part of) D and (ii) evaluating it on (another part of) D using a chosen performance index (accuracy, AUC, MAE, …).

Formally, we define the **learning-effectiveness function**:

$$f_{\text{learn-effect}} : \mathcal{L}_{X \to Y} \times \mathcal{P}^*(X \times Y) \to \mathbb{R}$$

where $\mathcal{L}_{X \to Y}$ is the set of learning techniques:

- $\mathcal{L}_{X \to Y} = \mathcal{F}_{\mathcal{P}^*(X \times Y) \to \mathcal{F}_{X \to Y}}$, or
- $\mathcal{L}_{X \to Y} = \mathcal{F}_{\mathcal{P}^*(X \times Y) \to M} \times \mathcal{F}_{X \times M \to Y}$ (when the model space M is explicit).

The dataset used for testing should be **unseen** with respect to the model m. This ensures that the evaluation measures the model's ability to **generalise** to new data:

- **Same dataset (training = testing)**:
  The simplest (**but usually wrong!**) approach is to train and test on the same dataset:
  $$D_{\text{learn}} = D_{\text{test}} = D.$$

  Warning: *Training accuracy is not a generalisation estimate*

  Using D both for learning and assessment does **not** measure generalisation. A learning technique can overfit D (learn noise and artefacts) resulting in high training performance even when test performance is poor.

  This approach can be used as a quick *sanity check*, but should not be used to estimate performances.

- **Static train/test split**:
  A standard alternative is to split the dataset once:
  $$D_{\text{learn}} \cup D_{\text{test}} = D, \qquad D_{\text{learn}} \cap D_{\text{test}} = \emptyset, \qquad |D_{\text{learn}}| \approx r|D|,$$
  where $r \in (0,1)$ is a chosen fraction (typical values are $r = 0.7$ or $r = 0.8$).

  Definition: *Static train/test division*

  – **Effectiveness:** generalisation is assessed (test set is unseen);
  – **Efficiency:** learning is executed only once;
  – **Weakness:** the estimate depends on the particular split (it may be lucky or unlucky).

Observation: *On the meaning of "unseen"*

D_{test}, with respect to the model m, is **unseen** data because it has not been used for learning. Assessing m on unseen data answers the following questions:

- To what degree does the model generalise **beyond examples**?
- Does the model work well on **new data**?

In practice, D_{test} and D_{learn} are obtained from a dataset D that is collected all at once; hence D_{test} might represent future data only roughly.

- **Repeated random train/test split**:
  To reduce the dependency on a single random split, the procedure is repeated k times:

$$(D_{\text{learn}}^{(j)}, D_{\text{test}}^{(j)}) \quad \text{for } j = 1, \ldots, k,$$

  and average the resulting performance values. Generalisation is assessed, and robustness w.r.t. the split increases; the cost is that learning is executed k times (proportional to k). Common values: $k = 5$ or $k = 10$. This produces not only an average performance but also a notion of variability, which is useful when comparing techniques.

- **Cross-validation (CV)**:
  In k-fold cross-validation, the dataset is partitioned into k disjoint folds of equal size:

$$D = F_1 \cup \cdots \cup F_k, \qquad F_i \cap F_j = \emptyset \ \ (i \neq j).$$

  For each fold j, the model is trained on the union of all other folds and evaluated on F_j:

$$D_{\text{test}}^{(j)} = F_j, \qquad D_{\text{learn}}^{(j)} = D \setminus F_j.$$

Definition: *k-fold Cross-Validation*

- **Effectiveness:** generalisation is assessed; robustness is usually better than a single split;
- **Efficiency:** learning is executed k times (cost $\propto k$);
- **Common choices:** $k = 5$ or $k = 10$.

  An advantage of CV is that every example is used for testing once and for training $k - 1$ times.

- **Leave-One-Out cross-validation (LOOCV)**:
  LOOCV is the extreme case of cross-validation where $k = |D|$: each fold contains exactly one observation. Generalisation is assessed with high robustness, but learning is executed $|D|$ times, which is often very expensive. In practice, LOOCV is mainly used when the dataset is very small and training is cheap.

Summary: effectiveness vs. efficiency

Method	Generalisation	Learning runs
Same dataset	no (poor)	1
Static (r)	yes, no robustness	1
Repeated random (k)	yes, robust	k
CV (k-fold)	yes, robust	k
LOOCV	yes, robust	$\|D\|$

Table 2.1: Effectiveness and efficiency of train/test division methods.

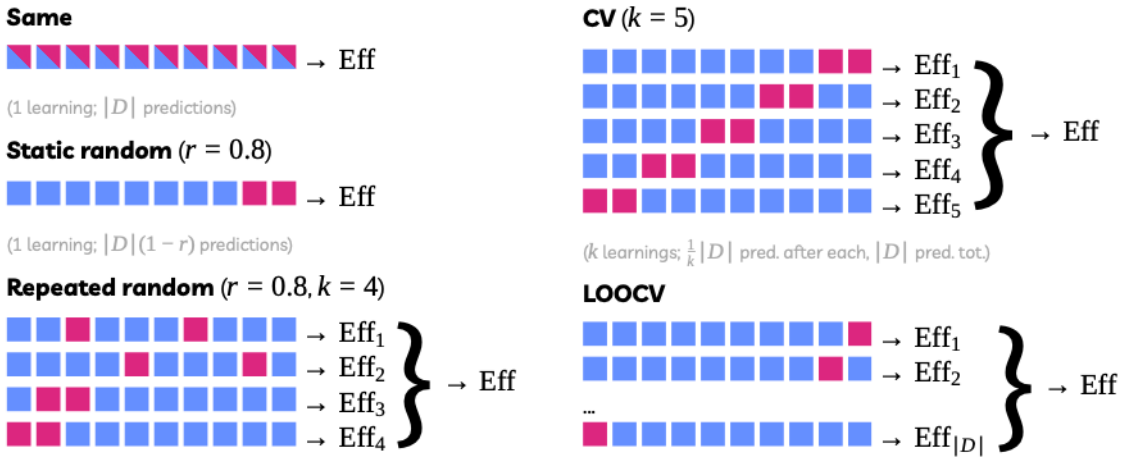


Figure 2.7: Train/Test division methods (conceptual view).

Increasing robustness (repeated splits, CV, LOOCV) improves the **quality of the estimate** of generalisation by reducing variance in the assessment, but requires training more models and thus reduces **computational efficiency**. This presents a fundamental **trade-off**: more reliable estimates come at the cost of increased computational resources. The choice of method should balance the need for accurate generalisation against available computational budget and dataset size.

Observation: ***Assessment vs. Reality***

Even when assessment on D_{test} is high, the model may fail on actual unseen or future data. Two main causes:

- D was not **representative** of the real system (low coverage, or data is old);
- D is not **informative** w.r.t. s (data is noisy or y in D doesn't depend on x).

Mean and standard deviation of performance

Repeated splits and cross-validation naturally produce a collection of performance values $v_1, v_2, \ldots, v_k$, where each v_j is computed on a different test set. It is then useful to report:

$$\mu_v = \frac{1}{k}\sum_{j=1}^{k} v_j, \qquad \sigma_v = \sqrt{\frac{1}{k}\sum_{j=1}^{k}(v_j - \mu_v)^2}.$$

- μ_v indicates the average effectiveness of the learning technique;
- σ_v indicates sensitivity to the specific training set (high σ_v signals instability or high variance).

Comparing learning techniques

When comparing two learning techniques, one can compare:

- a single number (e.g. μ_{AUC});
- both mean and variability (e.g. $\mu_{\text{AUC}} \pm \sigma_{\text{AUC}}$);
- the full distribution of the values via **boxplots**.

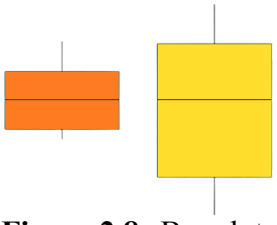


Figure 2.8: Boxplots

Tip: ***Boxplots as a practical comparison tool***

Boxplots are useful because they make differences in *central tendency* (median/mean) and in *dispersion* (variability across folds/splits) visible at a glance.

Warning: *Beyond boxplots: statistical significance*

When the difference between two techniques is small compared to the variability across folds, it may be unclear whether one method is truly better or whether the observed difference is attributable to noise. In such cases, a statistical significance test (e.g. Wilcoxon, Friedman) allows one to support the comparison. Such tests return a p-**value** $p \in [0,1]$: if $p < \alpha$ (typically $\alpha = 0.05$), one concludes that there is a statistically significant difference.

2.2 Tree-based Learning Techniques

This section introduces tree-based learning techniques with a motivating example: replacing a human attendant at a carousel with an automatic decision system.

2.2.1 The Decision Tree Model

Consider an amusement park where an attendant decides who can ride the carousel. The goal is to replace the attendant with a robotic gate. One observes the attendant's behaviour and collects datas $D = \{(x^{(i)}, y^{(i)})\}$ where x encodes the person approaching (`height` in cm, `age` in years) and y is the decision (`can ride`, `cannot ride`). Hence $X = \mathbb{R}^+ \times \mathbb{R}^+$ and $Y = \{\text{pos}, \text{neg}\}$.
By exploring the data, we can notice that the decision logic can be represented by a series of checks ("if age ≤ 10 then check height…"). This hierarchical structure of conditions is a **decision tree**.

Definition: *Decision tree*

A ***decision tree*** is a model $m \in T_{p,Y}$ defined over the input space X and output space Y.
It consists of a binary tree where:

- **Non-terminal nodes** (branch nodes) hold a pair (j, τ), where $j \in \{1, \ldots, p\}$ is the index of the variable to check and $\tau \in \mathbb{R}$ is the threshold;
- **Terminal nodes** (leaf nodes) hold a prediction $y \in Y$ (or a probability distribution over Y).

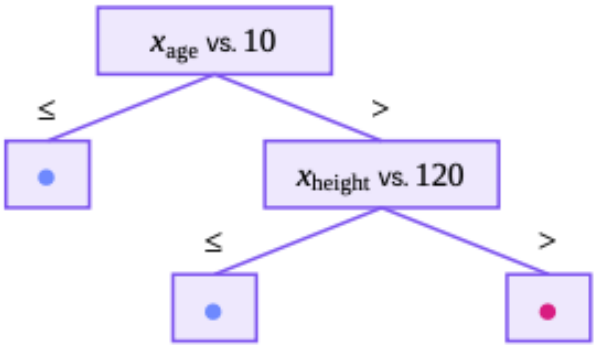


Figure 2.9: Decision tree example: branch nodes hold (variable, thr) pairs; leaves hold class labels.

In a compact representation, a tree t is either a leaf $[l]$ or a branch node $[(j, \tau); t_{\text{left}}; t_{\text{right}}]$, where $l \in L = (\{1, \ldots, p\} \times \mathbb{R}) \cup Y$ and $t_{\text{left}}, t_{\text{right}} \in T_L \cup \{\emptyset\}$.

$$t = [(1, 10); [\bullet]; [(2, 120); [\bullet]; [\bullet]]]$$

Figure 2.10: Compact representation of the decision tree.

The prediction function $f'_{\text{predict}}(x,t)$ works recursively:

Algorithm 1 Templated f'_{predict}

```
1:  function predict(x, t)
2:      if ¬ has-children(t) then          ▷ true iff t is not a leaf (i.e., t is a terminal node)
3:          return label(t)                ▷ return the class label y ∈ Y
4:      else                               ▷ t is a branch node
5:          (j, τ) ← label(t)              ▷ extract variable index and threshold
6:          if x[j] ≤ τ then
7:              return predict(x, left-child-of(t))    ▷ recurse on left subtree
8:          else
9:              return predict(x, right-child-of(t))   ▷ recurse on right subtree
10:         end if
11:     end if
12: end function
```

This model partitions the input space X into axis-parallel hyper-rectangles, assigning a constant value (class or mean) to each region.

2.2.2 Learning Decision Trees

The search space of all trees is too large to enumerate. Instead, one uses a greedy heuristic.

Recursive binary splitting

The learning function $f'_{\text{learn}}(D)$ builds the tree using a top-down (divide-et-impera) approach known as **recursive binary splitting**. The algorithm proceeds as follows:

1. Until a stopping criterion is satisfied (e.g., all labels are identical, or the node is too small);
2. Find a split (i.e., a variable j and threshold τ) that best separates the data;
3. Recursively apply steps 1-2 to each of the two resulting regions.

When recursion stops, the node becomes a leaf labelled with the **most frequent class** in that region.

Algorithm 2 Templated f'_{learn}

```
1:  function learn({(x^(i), y^(i))}_i)
2:      if should-stop({y^(i)}_i) then
3:          y* ← arg max_{y∈Y} Σ_i 1(y^(i) = y)    ▷ y* is the most frequent class
4:          return node-from(y*, ∅, ∅)
5:      else                                       ▷ hence r is a branch node
6:          (j, τ) ← find-best-branch({(x^(i), y^(i))}_i)
7:          t ← node-from(
8:                  (j, τ),
9:                  learn({(x^(i), y^(i))}_i|_{x_j^(i) ≤ τ}),    ▷ recursion
10:                 learn({(x^(i), y^(i))}_i|_{x_j^(i) > τ}))    ▷ recursion
11:         return t
12:     end if
13: end function
```

Finding the best branch

To find the best split, we iterate through all variables j and all possible thresholds τ. For each feature j, we only need to consider the midpoints between consecutive values of $(x_j^{(i)})$.

Intuitively:

- Consider all variables and all threshold values;
- Choose the pair (var, thr) that results in the lowest rate of **misclassified** examples.

Each split is evaluated using an **impurity measure**. For a dataset D (or node t), let p_y denote the frequency of class y in D. Common impurity measures are:

- **Classification error**: $1 - \max_y p_y$;
- **Gini index**: $\sum_y p_y(1-p_y) = 1 - \sum_y p_y^2$;
- **Cross-entropy**: $-\sum_y p_y \log_2(p_y)$.

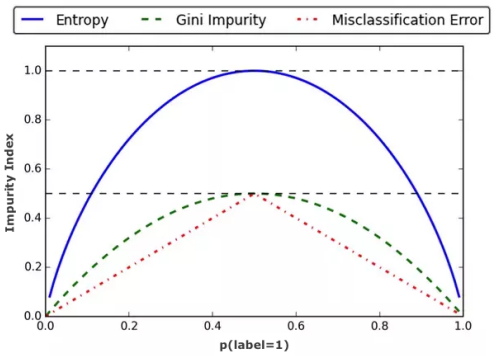


Figure 2.11: Gini, entropy and error.

Algorithm 3 find-best-branch

function find-best-branch($\{(x^{(i)}, y^{(i)})\}_i$)
  $(j^*, \tau^*) \leftarrow \arg\min_{j,\tau} \Big(\text{impurity}(\{y^{(i)}\}_i|_{x_j^{(i)} \le \tau}) + \text{impurity}(\{y^{(i)}\}_i|_{x_j^{(i)} > \tau}) \Big)$
  return (j^*, τ^*)
end function

function error($\{y^{(i)}\}_i$) ▷ the error of the dummy classifier on $\{y^{(i)}\}$
  $y^* \leftarrow \arg\max_y \sum_i \mathbf{1}(y^{(i)} = y)$ ▷ y^* is the most frequent class
  return $\frac{1}{n} \sum_i \mathbf{1}(y^{(i)} \neq y^*)$
end function

This approach is **greedy**, since it tries to obtain the maximum result (finding the branch), with the minimum effort (using just two dummy classifiers; later on): in practice, it makes this learning technique **computationally light**!

Tip: ***Gini vs. entropy***

Gini and cross-entropy are strictly convex and smoother than classification error, making them more sensitive to changes in node purity. Gini is generally preferred as it is computationally cheaper (no logarithms).

Stopping criteria and complexity

If the tree grows indefinitely, it will eventually isolate every training example. This leads to maximal complexity. Stopping criteria are required:

- **Data size**: stop if $|D| \le n_{\min}$ or $\text{error}(\{y^{(i)}\}_i) = 0$, it is the most common parameter;
- **Depth**: stop if the node depth exceeds a maximum depth τ_d;
- **Impurity**: stop if the impurity is below a threshold τ_ε.

For the same dataset, in general, parameters impact the tree size as follows:

- the lower $n_{\min}$, the **larger** the tree;
- the greater τ_d, the **larger** the tree;
- the lower τ_ε, the **larger** the tree.

2.2.3 Tree learning with probabilities

In standard formulation, leaves hold a single class label $y \in Y$. Alternatively, they can hold **discrete probability distributions** over Y, corresponding to the frequencies of each class in the leaf.

Definition: *Tree learning with probability*

The node labels are $L = (\{1,\dots,p\} \times \mathbb{R}) \cup \mathcal{P}_Y$: branch nodes hold (j,τ); terminal nodes hold discrete distributions $p \in \mathcal{P}_Y$ instead of a single class.
The learning and prediction functions are:

$$f'_{\text{learn}} : \mathcal{P}^*(X_1 \times \cdots \times X_p \times Y) \to T_{(\{1,\dots,p\}\times\mathbb{R})\cup\mathcal{P}_Y},$$

$$f''_{\text{predict}} : X_1 \times \cdots \times X_p \times T_{(\{1,\dots,p\}\times\mathbb{R})\cup\mathcal{P}_Y} \to \mathcal{P}_Y.$$

A hard prediction is obtained as $\hat{y} = \arg\max_{y\in Y} p(y)$ given $p = f''_{\text{predict}}(x,m)$.

Learning: f'_{learn} with probability

The only change w.r.t. the standard algorithm is the treatment of leaves. When a stopping condition is met, instead of assigning the most frequent class, we assign the **empirical frequency distribution**:

$$p = y \mapsto \text{Fr}(y, \{y^{(i)}\}_i),$$

where $\text{Fr}(y, \{y^{(i)}\}_i)$ is the frequency of class y in the labels at that node. The leaf then returns node-from$(p,\emptyset,\emptyset)$ instead of node-from$(y^*,\emptyset,\emptyset)$.

Example: *Before vs. after*

Consider a leaf with 5 observations: 3 of class •, 1 of class ○, 1 of class ⋆.
Without probability: the leaf returns the single label • (the majority class).
With probability: the leaf returns $p = (\frac{3}{5}\bullet, \frac{1}{5}\circ, \frac{1}{5}\star)$, i.e. empirical frequencies of each class.

Prediction: f'_{predict} vs. f''_{predict}

Two prediction functions can be defined:

- $f'_{\text{predict}} : X \times M \to Y$ (hard prediction): at a leaf, compute $\hat{y} = \arg\max_{y\in Y} p(y)$, and return $\hat{y}$;
- $f''_{\text{predict}} : X \times M \to \mathcal{P}_Y$ (soft prediction): at a leaf, return the distribution p directly.

In practice, ML libraries often provide both $\hat{y}$ and p from a single tree traversal.

2.2.4 Model complexity and generalisation

The learned tree may be too large w.r.t. the real system. Almost every learning technique has at least one parameter affecting the maximum complexity of the learnable models, often called **flexibility**: more flexibility → more complex models; less flexibility → simpler models.
The model strives for maximum accuracy on the training set, but **noise** in the data makes a perfect score impossible (and undesirable). The goal is to model the real system s, not s plus noise.

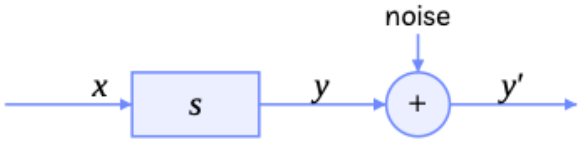


Figure 2.12: Noise in the data: the goal is to model s, not s + noise.

Overfitting and underfitting

Definition: *Overfitting*

Given a **noisy dataset** and **large complexity** (a flexibility parameter set high), the learning technique may **fit the noisy data** instead of the real system s; **overfitting** occurs.

- **Overfitting**: the model fits the training data *too well*, capturing noise and artefacts rather than the underlying system s. This happens if the model is too complex (high flexibility, e.g., $n_{\min} = 1$).
- **Underfitting**: the model is too simple to capture the structure of s. This happens if flexibility is too low (e.g., $n_{\min} \to \infty$, resulting in a single root node).

Bias and variance

These concepts correspond to statistical properties of the learning technique:

- **High bias (underfitting)**: the technique systematically errs in the same way regardless of the training data (e.g., a dummy classifier always predicts the majority class). It is blind to the data.
- **High variance (overfitting)**: the model changes drastically with small changes in training data. The technique is too sensitive to the specific dataset instance.

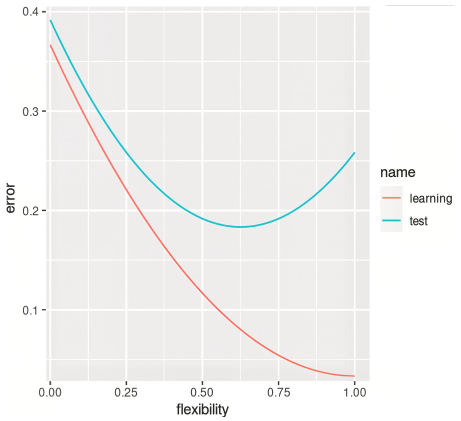


Figure 2.13: Bias-variance trade-off.

Example: *Practical procedure for choosing flexibility*

1. Consider different values of the flexibility parameter;
2. Choose a suitable effectiveness index and learning/testing division method;
3. For each value, learn a model and compute performance index on both train and test sets.

2.2.5 Hyperparameter tuning

Parameters like $n_{\min}$ or τ_d are not learned from data but set by the designer. These are called **hyperparameters**. Choosing the right hyperparameters is crucial for balancing bias and variance.

Definition: *Hyperparameter tuning*

Given a learning technique with h parameters $p_1, ..., p_h$ and $p_j \in P_j$ for each j, ***hyperparameters*** are parameters whose values are not learned from data but are set by the designer before training. ***Hyperparameter tuning*** is the task of finding the tuple of hyperparameters $(p_1^*, ..., p_h^*)$ that maximises effectiveness:

$$(p_1^*, ..., p_h^*) = \underset{(p_1,...,p_h) \in P_1 \times \cdots \times P_h}{\arg\max} f_{\text{effectiveness}}(p_1, ..., p_h).$$

The most common method is **grid search**:

1. For $j = 1, ..., h$, the hyperparameter designer chooses a finite set $P'_j \subseteq P_j$ of candidate Values;
2. Construct the cartesian product $P'_1 \times \cdots \times P'_h$ (the **grid**);
3. For each configuration $(p_1, ..., p_h)$, estimate effectiveness using a robust method (e.g. k-fold CV):

$$(p_1^*, ..., p_h^*) = \underset{(p_1,...,p_h) \in P'_1 \times \cdots \times P'_h}{\arg\max} \hat{f}_{\text{effectiveness}}(p_1, ..., p_h);$$

4. Select the configuration with the best estimated performance.

The choice of P'_j sets is crucial: too small may miss good values; too large is expensive.

Hyperparameter-free learning

The combination of (grid search + learning technique) can be viewed as a new, **hyperparameter-free** learning technique. It is more effective (it adapts to the problem) but computationally more expensive (it trains many models). The chosen values may overfit the specific dataset used for tuning, thus lacking generalisation.

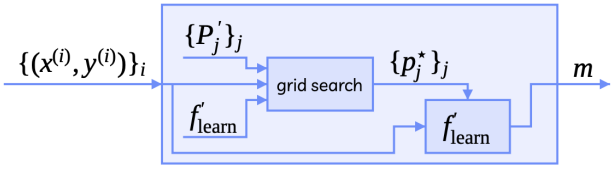


Figure 2.14: Hyperparameter-free learning: grid search wraps any learning technique.

2.2.6 Categorical independent variables and regression

For categorical variables, the split is defined by a variable x_j and a subset $X'_j \subset X_j$: observations with $x_j^{(i)} \in X'_j$ go to one child, those with $x_j^{(i)} \notin X'_j$ to the other.

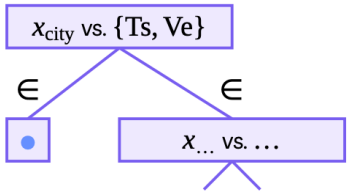


Figure 2.15: Tree with categorical variables: split on $x_j \in X'_j$ vs. $x_j \notin X'_j$.

For a given **categorical variable** $x_j \in X_j$, one chooses $X'_j \subset X_j$ such that:

$$X'_j = \underset{X'_j \in \mathcal{P}(X_j)}{\arg\min} \left(f_{\text{impurity}}(\{y^{(i)}\}_i \,|_{x_j^{(i)} \in X'_j}) + f_{\text{impurity}}(\{y^{(i)}\}_i \,|_{x_j^{(i)} \notin X'_j}) \right).$$

In a **regression tree**, terminal nodes hold numerical values instead of classes. Impurity and classification error no longer apply; the split criterion and stopping condition must be adapted. The best branch minimises the residual sum of squares, $\text{RSS}(\{y^{(i)}\}_i) = \sum_i (y^{(i)} - \hat{y}^{(i)})^2$:

$$(j^*, \tau^*) = \underset{j,\tau}{\arg\min} \left(\text{RSS}(\{y^{(i)}\}_i \,|_{x_j^{(i)} \leq \tau}) + \text{RSS}(\{y^{(i)}\}_i \,|_{x_j^{(i)} > \tau}) \right).$$

The stopping criterion uses RSS instead of error; recursion stops when $\text{RSS} = 0$ or when $n \leq n_{\min}$. Note that $\text{RSS} = 0$ is much rarer than error $= 0$ in classification trees.

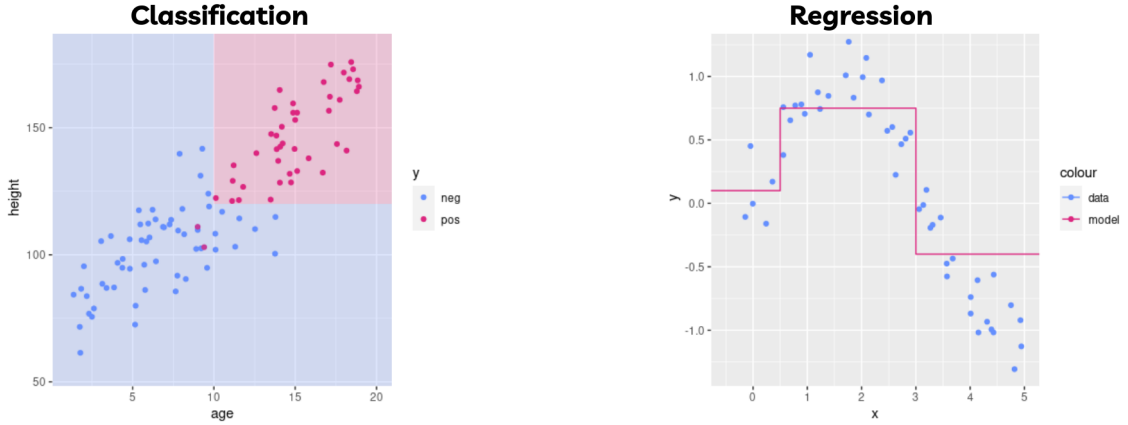


Figure 2.16: Classification tree vs. regression tree.

2.3 Ensemble methods: Tree Bagging and Random Forest

Single decision trees, while easy to interpret, suffer from a significant drawback: high variance.
If we learn a regression tree with **low flexibility** (e.g., small depth):

- the model will not capture the system behavior;
- it will **underfit** the data and the system (high bias).

If we learn a regression tree with **high flexibility** (e.g., fully grown):

- the model will likely better capture the system behavior, but…
- it will model also some noise;
- it will **overfit** the data (high variance).

Ensemble methods are techniques that combine multiple **weak learners** (models that perform slightly better than random guessing, in this case decision trees) to create a stronger, more robust predictor. The goal is to combine the power of high-flexibility models (low bias) while reducing their variance. Ideally, we want to average out the noise by combining multiple models.

Definition: ***Wisdom of the crowds theorem***

A collective opinion may be better than a single expert's opinion.

The aggregation of models works only if:

- we have **many opinions** → we learn many trees;
- the opinions are **independent** → each tree is learned on a different dataset;
- we have a way to **aggregate** them → we average predictions or take a majority vote.

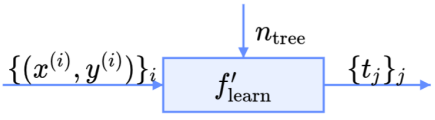
2.3.1 Tree Bagging

To satisfy the independence requirement, we need different learning sets. Since we usually only have one dataset D, we simulate having multiple datasets using **bootstrapping**.
Bootstrap aggregating, or **Bagging**, involves generating B different training datasets by sampling with replacement from the original dataset D. We then train a tree on each bootstrapped dataset and aggregate their predictions.

Definition: ***Tree Bagging***

- **Learning:** The model is an ensemble (a "bag") of n_{tree} trees.
  - Each tree is trained on a bootstrap sample of the data.
  - The learning process is **non-deterministic** due to the random sampling.
- **Predict:**
  - **Classification:** Majority vote among the trees (return the most frequent class).
  - **Regression:** Average of the predictions of individual trees. Ideally, the standard deviation σ across trees is also returned as a measure of **uncertainty**.

The hyperparameter n_{tree} is **not** a flexibility parameter in the traditional sense. Unlike other hyperparameters that control model complexity, increasing n_{tree} does not cause overfitting; it simply stabilizes the error convergence.



With too few trees, the aggregation exhibits high variance, leading to unstable predictions. Conversely, using too many trees yields diminishing returns in predictive performance while incurring higher computational costs. Therefore, choosing n_{tree} is primarily a trade-off between **efficiency** and stability, rather than a bias-variance consideration.

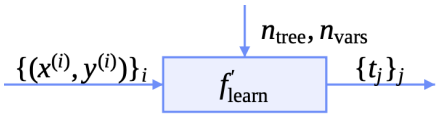
2.3.2 Random Forest

While bagging reduces variance, the trees in a bagged ensemble can still be highly **correlated**. If there is one very strong predictor (feature) in the dataset, most trees will use it for the first split, making them look very similar. To further increase **independence**, we need to decorrelate the trees. We do this by injecting randomness not just in the *rows* (bootstrapping) but also in the *columns* (features).

Definition: ***Random Forest***

Random Forest is a learning technique that enhances Tree Bagging by introducing a random restriction on the features available for splitting.
When building each tree, at **each split**, the algorithm is only allowed to consider a random subset of n_{vars} features chosen from the total p features.

- **Random:** because there are two sources of randomness (bootstrap samples and feature sampling).
- **Forest:** because the model is a collection of trees.



The model is a bag of n_{tree} trees. The parameter n_{vars} (or m, in literature) controls the correlation between trees:

- If $n_{\text{vars}} = p$, Random Forest matches Bagging.
- If n_{vars} is small, trees are very diverse (decorrelated).

Experimentally, n_{vars} does not strongly impact the tendency to overfit. Therefore, Random Forest is often considered practically **hyperparameter-free** because robust default values exist:

- Classification: $n_{\text{vars}} \approx \sqrt{p}$
- Regression: $n_{\text{vars}} \approx \frac{p}{3}$

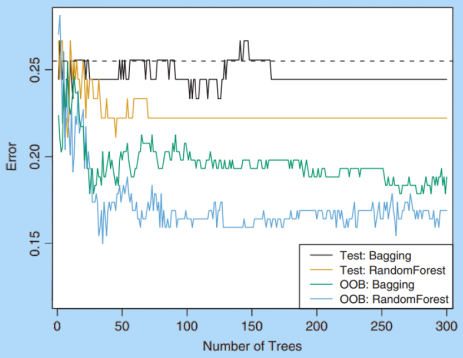
2.3.3 Out-Of-Bag (OOB) Assessment

Since each tree is trained on a bootstrap sample, it leaves out approximately one-third of the observations. These leftover observations are called **Out-Of-Bag (OOB)** samples for that specific tree. We can use these OOB to estimate the test error without requiring a separate validation set.

Definition: ***OOB Error***

The ***OOB error*** is an unbiased estimate of the generalization error computed during training:

1. For each observation $x^{(i)}$, identify the trees for which $x^{(i)}$ was OOB (not in the bootstrap).
2. Aggregate the predictions of only those trees for $x^{(i)}$.
3. Compute the error (accuracy/MSE) based on these aggregated predictions.



Warning: ***Interpretability***

Ensemble methods **lack of Interpretability**, since they consider different scenarios and use the computational power to find human-invisible patterns in the data.

2.3.4 Variable Importance

One downside of ensemble methods is the loss of interpretability. We can no longer look at a single tree to understand the decision process. To mitigate this, we calculate **variable importance** metrics to understand which features drive the predictions.

Method 1: Mean Decrease Impurity (MDI)

During training, every time a node splits on variable x_j, the impurity (Gini or Entropy) decreases. We can sum these decreases for each variable across all trees.

- **Pros:** Fast to compute (done during training).
- **Cons:** Biased towards numerical variables and categorical variables with high cardinality (many categories).

Method 2: Mean Decrease Accuracy (Permutation Importance)

This method measures importance by breaking the association between a feature and the target.

1. For each tree t and its OOB data:
   (a) Record the accuracy of t on the OOB data.
   (b) **Shuffle** the values of variable x_j in the OOB data (breaking the relationship with y).
   (c) Measure accuracy again on the shuffled data.
   (d) The **decrease in accuracy** is the importance of x_j for tree t.
2. Average the decreases across all trees.

Tip: ***Rationale***

If a variable is important, shuffling it should drastically effectively destroy the model's ability to predict, leading to a large drop in accuracy. If shuffling makes no difference, the variable was not used or not useful.

Method 3: Feature Ablation

A more general (but computationally expensive) approach applicable to any model:

1. Measure effectiveness of the model on the full dataset D.
2. For each variable x_j:
   (a) Create dataset D' by **removing** x_j.
   (b) Retrain and measure effectiveness on D'.
   (c) The decrease in effectiveness represents the importance of x_j.

Definition: ***No Free Lunch Theorem***

The ***No Free Lunch Theorem*** states that no single machine learning algorithm is universally better than all others across all possible domains.
While Random Forests are robust and effective in many scenarios, there are specific problems where other techniques (like linear models or neural networks) may perform better.

2.4 Support Vector Machines

Previous learning techniques, such as Decision Trees, divide the input space into hyper-rectangles with boundaries parallel to the axes. In many scenarios, they can be ineffective for datasets where the optimal separation between classes is linear but **oblique** (not axis-aligned), or follows a more complex boundary. In such cases, a tree would require many splits to approximate a single diagonal line, leading to complex models and potential overfitting.

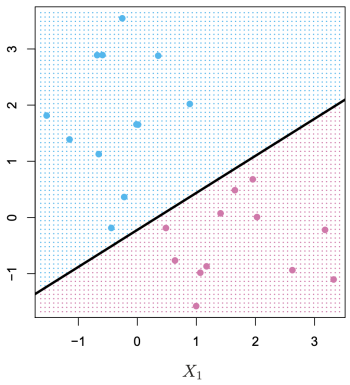


Figure 2.17: A dataset where an oblique decision boundary is better than an axis-parallel one.

2.4.1 The Separating Hyperplane

Support Vector Machines (SVMs) are intended for the **binary classification**. The fundamental idea is to find a **hyperplane** that separates the two classes in the feature space $X = \mathbb{R}^p$.

Definition: *Hyperplane*

In a p-dimensional space, a ***hyperplane*** is a flat affine subspace of dimension $p-1$. It is defined by the equation:

$$f(x) = \beta_0 + \beta^\top x = 0$$

where $\beta \in \mathbb{R}^p$ is the normal vector (determining the orientation) and β_0 is the intercept. The term "separating" indicates that the hyperplane divides the space into two parts; "hyperplane" refers to the geometry ($p = 1$: threshold; $p = 2$: line; $p = 3$: plane; $p > 3$: hyperplane).

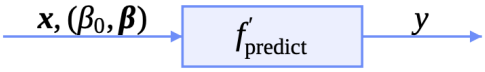


Figure 2.18: SVM f'_{predict}: classify as Pos if $\beta_0 + \beta^\top x \geq 0$, else Neg.

The hyperplane divides the space into two parts. We use it as a prediction function f'_{predict}:

- If $f(x) > 0$, we classify x as **Positive**.
- If $f(x) < 0$, we classify x as **Negative**.

Assumptions: $Y = \{\text{Pos}, \text{Neg}\}$ (binary classification only); $X = \mathbb{R}^p$ (numerical inputs only). Prediction is computationally very fast: just p multiplications and sums.

The **magnitude** of $f(x)$ tells us the (signed) distance of the point from the hyperplane. The larger $|f(x)|$, the further the point is from the boundary, and the more **confident** we are in the prediction. Note that $f(x)$ is not bounded in $[0, 1]$, so it cannot be interpreted as a probability.

2.4.2 The Maximal Margin Classifier

If the data is linearly separable, infinitely many hyperplanes separate the classes. Which one is the best? Intuitively, we want the hyperplane that passes right in the middle of the two classes, maximizing the distance to the nearest points on both sides.
This creates the widest possible "safety margin."

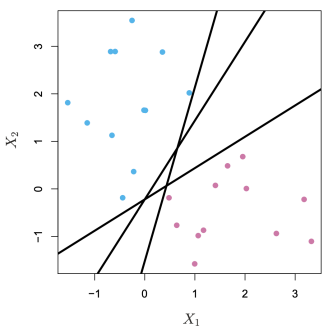


Figure 2.19: Choosing the best separating hyperplane.

Definition: ***Maximal Margin Hyperplane***

The ***margin*** is the perpendicular distance from the separating hyperplane to the closest training observation.
The ***Maximal Margin Hyperplane*** is the specific separating hyperplane that maximizes this margin. The corresponding classifier is called the ***Maximal Margin Classifier (MMC)***.

The training observations that lie exactly on the margin boundary are called **Support Vectors**. They "support" the hyperplane; if we move any point that is not a support vector (within limits), the separating hyperplane stays the same. If we move a support vector, the model changes.

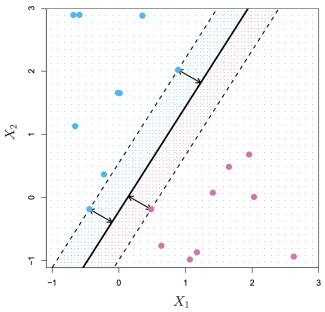


Figure 2.20: Maximal Margin Hyperplane. Points lying on the dashed lines are Support Vectors.

Learning as Optimization

Finding the Maximal Margin Hyperplane is a constrained optimization problem. Assume by convention that Pos $= +1$ and Neg $= -1$. We look for the coefficients β that maximize the margin M, subject to the constraint that every point is correctly classified and at least distance M from the hyperplane:

$$\begin{aligned} &\text{maximize } M \\ &\text{subject to } \sum_{j=1}^{p} \beta_j^2 = \beta^\top \beta = 1 \\ &y^{(i)}(\beta_0 + \beta^\top x^{(i)}) \geq M, \quad \forall i = 1, \ldots, n \end{aligned}$$

If $\beta^\top \beta = 1$, then $\beta_0 + \beta^\top x$ is the Euclidean distance of x from the hyperplane (with sign). The constraint holds with equality for support vectors and with strict inequality for the others. In practice, this is an easy optimization problem and solving it is fast.

Issues with Maximal Margin

The MMC is theoretically elegant but has two major practical issues:

1. **Sensitivity to noise (high variance):** The margin is determined solely by the support vectors. If we move a support vector slightly (e.g., by applying noise to some $x^{(i)}$), the separating hyperplane can shift dramatically; the model exhibits high sensitivity to small perturbations in the training data, leading to poor generalization.
2. **Non-separability:** If we apply noise to some label $y^{(i)}$, or if the classes are inherently overlapping, a separating hyperplane may not exist at all. In such cases, the optimization problem has no solution.

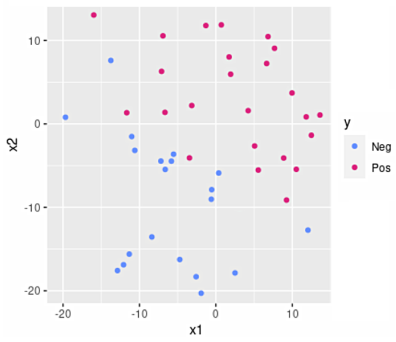


Figure 2.21: Noise in the data.

Tip: ***Applicability***

How did the tree cope with label noise? By tolerating some misclassifications on the learning data. Can we make the MMC tolerant too?

2.4.3 Soft Margin Classifier (Support Vector Classifier)

To make the model robust to noise and applicable to non-separable data, we introduce **tolerance**. We allow some points to violate the margin (or even be on the wrong side of the hyperplane). This is the **Soft Margin Classifier (SMC)**, also called **Support Vector Classifier**.
We introduce **slack variables** $\varepsilon^{(i)} \geq 0$ for each observation. There are two equivalent formulations.

Definition: *Soft Margin Classifier*

- **1st Formulation: Budget Constraint**
  - Maximize M
  - subject to $\beta^\top \beta = 1$
  - $y^{(i)}(\beta_0 + \beta^\top x^{(i)}) \geq M(1-\varepsilon^{(i)}) \quad \forall i$
  - $\varepsilon^{(i)} \geq 0 \quad \forall i$
  - $\sum_{i=1}^{n} \varepsilon^{(i)} \leq C$

  The parameter $C \in \mathbb{R}^+$ is a **budget of tolerance**. If $\varepsilon^{(i)} = 0$, the point is on the correct side of the margin; if $0 < \varepsilon^{(i)} \leq 1$, it is inside the margin but correctly classified; if $\varepsilon^{(i)} > 1$, it is misclassified.
- **2nd Formulation: Penalty Cost**
  - Maximize $M - c\sum_{i=1}^{n} \varepsilon^{(i)}$
  - subject to $\beta^\top \beta = 1$
  - $y^{(i)}(\beta_0 + \beta^\top x^{(i)}) \geq M(1-\varepsilon^{(i)}) \quad \forall i$
  - $\varepsilon^{(i)} \geq 0 \quad \forall i$

  The slack sum is unbounded but penalized in the objective. The parameter $c \in \mathbb{R}^+$ is a **weighting factor** balancing margin width vs. tolerance. Most ML libraries (e.g., scikit-learn, R) use this formulation or an equivalent such as $\min_\beta \left(\frac{1}{2}\|\beta\|^2 + C\sum_i \varepsilon^{(i)}\right)$.

The Parameter C (Bias-Variance Trade-off)

The two formulations assign *opposite* meanings to extreme values of the parameter.

- **1st formulation** (budget $\sum \varepsilon \leq C$):
  - $C = 0$: no tolerance $\Rightarrow$ Maximal Margin Classifier (high variance).
  - $C \to \infty$: infinite budget $\Rightarrow$ high bias.
- **2nd formulation** (penalty $M - c\sum \varepsilon$):
  - $c = 0$: no penalty for violations $\Rightarrow$ points inside the margin cost zero; the model is the same irrespective of data (high bias).
  - $c \to \infty$: infinite penalty $\Rightarrow$ maximum effort to put all points outside the margin; the margin is very sensitive to point positions (high variance). A model can always be learned.

Learnability: With the 2nd formulation, a model can always be learned from any dataset D. With the 1st formulation, there exists $c_{\text{learnable}} \geq 0$ such that if $C < c_{\text{learnable}}$, no model can be learned ($c_{\text{learnable}} = 0$ when the data is linearly separable).

Importance of Scaling

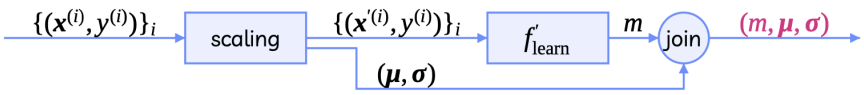
The coefficients β_j depend on the scales of the variables. If $x_j \in [1.4, 2.1]$ (e.g., height in metres) and $x_{j'} \in [20000, 50000]$ (e.g., income in euros), then β_j and $\beta_{j'}$ will differ greatly, making the model sensitive to noise.

Tip: *Pre-processing for SVM*

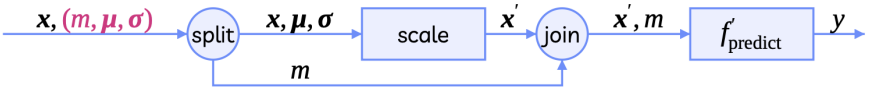
You **must** rescale numerical variables before applying MMC, SMC, or SVM. Options:

- **Min-max scaling:** $x'_j = \frac{x_j - \min_{i'} x_j^{(i')}}{\max_{i'} x_j^{(i')} - \min_{i'} x_j^{(i')}}$
- **Standardization:** $x'_j = \frac{1}{\sigma_j}(x_j - \mu_j)$ (preferred, more robust to outliers)

The scaling parameters (μ, σ or min, max) become part of the model, since scaling must be applied both in learning and prediction.



(a) Learning with standardization: scaling parameters are stored in the model.



(b) Prediction with standardization: apply the same scaling before f'_{predict}.

Figure 2.22: Standardization in learning and prediction phases.

2.4.4 Support Vector Machines (Kernels)

The Soft Margin Classifier yields a linear decision boundary. Many real-world problems are not linearly separable. We can extend linear classifiers by **enlarging the feature space**: transform inputs x into a higher-dimensional space via $\phi(x)$, find a linear separating hyperplane there, and project it back to obtain a non-linear boundary in the original space.

X_j	Y	RF	SVM
Numerical	Binary classification	✓	✓
Categorical	Binary classification	✓	×
Numerical + Categorical	Binary classification	✓	×
Numerical	Multiclass classification	✓	×
Categorical	Multiclass classification	✓	×
Numerical + Categorical	Multiclass classification	✓	×
Numerical	Regression	✓	×
Categorical	Regression	✓	×
Numerical + Categorical	Regression	✓	×

Table 2.2: Comparison of applicability: Random Forest (RF) vs Support Vector Machine (SVM).

Alternative Formulation and the Kernel Trick

The prediction function can be written in two equivalent forms:

$$f(x) = \beta_0 + \beta^\top x = \beta_0 + \sum_{j=1}^{p} \beta_j x_j \qquad \text{or} \qquad f(x) = \beta_0 + \sum_{i=1}^{n} \alpha^{(i)} \langle x, x^{(i)} \rangle$$

where $\langle x, x' \rangle = x^\top x'$ is the inner product. The $\alpha^{(i)}$ are determined by the optimization; $\alpha^{(i)} = 0$ for every $x^{(i)}$ that is *not* a support vector. Each $\alpha^{(i)}$ indicates the contribution of $x^{(i)}$ when classifying x. The second form generalizes to a **kernel function** $k : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha^{(i)} k(x, x^{(i)})$$

Definition: *Kernel*

The idea behind the kernel is to:

1. Transform the original space $X = \mathbb{R}^p$ into another space $X' = \mathbb{R}^q$ (possibly $q \gg p$) via $\phi : X \to X'$, and
2. Compute the inner product in the destination space: $k(x,x') = \phi(x)^\top \phi(x')$

hoping that a hyperplane in X' separates the points better than in X. This is the **kernel trick**. When using a kernel, the technique is called **Support Vector Machines (SVM)**.

Some common kernels are:

- **Linear kernel:** $k(x,x') = x^\top x'$. The most efficient (computationally cheapest); equivalent to the Soft Margin Classifier.
- **Polynomial kernel:** $k(x,x') = (1 + x^\top x')^d$. The degree d is a parameter.
- **Gaussian (RBF) kernel:** $k(x,x') = \exp(-\gamma \|x - x'\|^2)$. The most widely used; γ is a flexibility parameter. Also called radial basis function (RBF) or radial kernel.

Intuition of the Gaussian kernel: It maps x to a space where coordinates are (implicitly) the distances to relevant observations of the learning data. The decision boundary can smoothly follow any path, with some risk of overfitting. The larger γ, the faster $e^{-\gamma \|x-x'\|^2}$ goes to 0 with distance; high γ yields a more complex boundary.

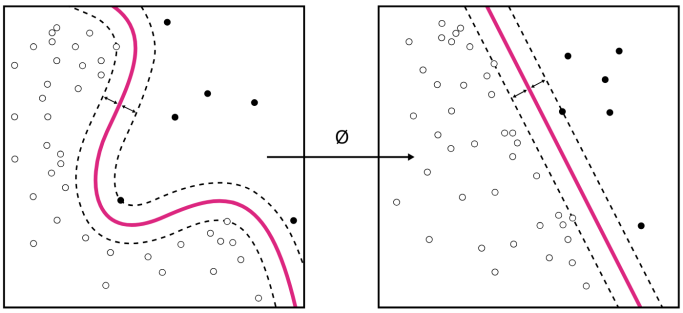


Figure 2.23: Gaussian kernel: the decision boundary is a smooth curve that can follow any path, with some risk of overfitting.

Observation: *SVM Summary*

Efficiency: very fast. **Effectiveness:** in general good with the Gaussian kernel; complex interactions between C and γ require careful parameter tuning. **Applicability:** Y binary only; X numerical only; models give a confidence (not a probability); two parameters (C and γ). **Interpretability:** few numbers, hardly interpretable; knowing the support vectors helps as a basic form of local explainability.

2.4.5 Applicability and Extensions

SVMs are designed for $Y \in \{\text{Pos}, \text{Neg}\}$ and $X \in \mathbb{R}^p$. we can extend to handle other settings.

Handling Categorical Variables

SVMs cannot interpret categorical levels directly. We must perform **encoding**:

- **One-hot encoding:** Replace a categorical variable $x_j \in \{x_{j,1}, \ldots, x_{j,k}\}$ with k binary variables $x_{h_1}, \ldots, x_{h_k} \in \{0,1\}$ such that $x_{h_\ell}^{(i)} = \mathbf{1}(x_j^{(i)} = x_{j,\ell})$. Each resulting binary variable is a **dummy variable**. This increases the number of predictors.
- **Ordinal encoding:** Assign integers to levels; only valid when there is an intrinsic order (e.g., Low, Medium, High).

Handling Multiclass Problems ($|Y| > 2$)

Since SVMs are binary classifiers, we decompose the problem:

- **One-vs-One (OvO):** Train a classifier for every pair of classes ($k(k-1)/2$ models). Classify by majority voting over the $\binom{k}{2}$ predictions.
- **One-vs-All (OvA) / One-vs-Rest:** Train one classifier per class, distinguishing it from all others (k models). Requires a base learner with confidence/probability output; assign the class whose classifier gives the highest confidence.

Warning: *Missing Values*

SVMs cannot handle missing values (they cannot compute $x^\top x^{(i)}$). You must:

- Drop observations (if few) or predictors (if mostly empty).
- **Impute** the missing values (e.g., with mean, median, or KNN imputation) before training.

2.5 Naive Bayes

This family of learning techniques is based on **Bayes' Theorem**. It classifies data calculating the probability of a data point belonging to a specific class given its features.

Definition: *Bayes' Theorem*

Recall the fundamental theorem of probability theory:

$$\Pr(A \mid B) = \frac{\Pr(B \mid A) \cdot \Pr(A)}{\Pr(B)}$$

In the context of supervised learning, we want to find the most probable class y given an observation vector $x = (x_1, \ldots, x_p)$. Let $A = y$ (the class) and $B = x$ (the evidence). The equation becomes:

$$\Pr(y \mid x_1, \ldots, x_p) = \frac{\Pr(x_1, \ldots, x_p \mid y) \cdot \Pr(y)}{\Pr(x_1, \ldots, x_p)}$$

where:

- $\Pr(y \mid x)$: **Posterior probability**. The probability of class y given the observed features. This is what we want to calculate.
- $\Pr(y)$: **Prior probability**. The probability of class y being seen in general (frequency in dataset).
- $\Pr(x \mid y)$: **Likelihood**. The probability of observing features x given that the class is y.
- $\Pr(x)$: **Evidence**. The probability of observing features x. Since this is constant for all classes y, it serves only as a normalization factor and can often be ignored during comparison.

2.5.1 The "Naive" Assumption

Estimating the full likelihood $\Pr(x_1, \ldots, x_p \mid y)$ is computationally expensive and requires a massive amount of data because it involves the joint probability of all features appearing together.

To make this feasible, Naive Bayes makes a strong (and "naive") assumption: **all independent variables (features) are independent of each other**, given the class label. Mathematically, this simplifies the likelihood into a product of individual probabilities:

$$\Pr(x_1, \ldots, x_p \mid y) \approx \prod_{j=1}^{p} \Pr(x_j \mid y)$$

The classification rule becomes:

$$\hat{y} = \arg\max_{y \in Y} \left(\Pr(y) \cdot \prod_{j=1}^{p} \Pr(x_j \mid y) \right)$$

Definition: ***Naive Bayes Learning Technique***

- **Learning Phase (f'_{learn})**: The model computes and stores the **prior probabilities** $\Pr(y)$ for each class and the **conditional probabilities** $\Pr(x_j \mid y)$ for every feature value given every class.
- **Prediction Phase (f'_{predict})**: For a new observation x, computes the product of the priors and conditional probabilities for each class, then selects the class with the highest score.

Model Complexity

The model is simply a data structure holding the probability tables.

- k values for priors $\Pr(y)$.
- $\sum_{j=1}^{p} k \cdot h_j$ values for conditional probabilities (where h_j is the # of possible values of feature j).

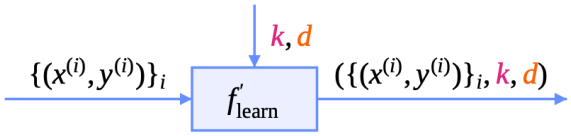
This makes Naive Bayes extremely **space-efficient** and **fast** for both training and prediction.

Warning: ***The Zero-Frequency Problem***

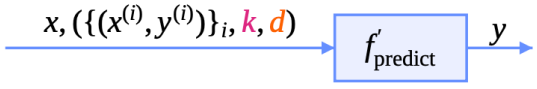
If a specific feature value x_j never appears with class y in the training set, $\Pr(x_j \mid y) = 0$. Because of the product, the entire probability becomes 0, erasing all other information. To fix this, we often use **Laplace smoothing** (adding a small count, usually +1, to every frequency).

2.6 k-Nearest Neighbors (kNN)

k-Nearest Neighbors (kNN) is a completely different approach based on the intuition that similar data points exist in close proximity in the feature space. It is an example of **Instance-based Learning** (or Lazy Learning).



(a) Learning: f'_{learn} stores the entire training dataset.



(b) Prediction: f'_{predict} uses stored data, k and d.

Figure 2.24: kNN stores the entire training dataset during learning and uses it directly for prediction.

The Algorithm

Unlike other methods, kNN does not "learn" a compressed model (like coefficients or trees).

- **Learning (f'_{learn})**: Simply stores the whole dataset D_{learn}. Effectively, the *model* is the data itself.
- **Prediction (f'_{predict})**:
  1. Computes the distance $d(x, x^{(i)})$ between the new point x and every point $x^{(i)}$ in the database.
  2. Selects the set I of the k points with the smallest distance.
  3. **Classification**: Returns the majority class among the neighbors (voting).
  4. **Regression**: Returns the average value of the neighbors: $\hat{y} = \frac{1}{k} \sum_{i \in I} y^{(i)}$.

Distance Metrics

The performance of kNN relies heavily on how "similarity" is defined. We need a distance metric $d : X \times X \to \mathbb{R}^+$. For $X = \mathbb{R}^p$, some common distance functions are:

- **Euclidean Distance** (L_2 norm): The straight-line distance.

$$d(x,x') = \sqrt{\sum_{j=1}^{p} (x_j - x'_j)^2}$$

- **Manhattan Distance** (L_1 norm): Sum of absolute differences (taxicab geometry).

$$d(x,x') = \sum_{j=1}^{p} |x_j - x'_j|$$

- **Minkowski Distance** (Generalization):

$$d(x,x') = \left(\sum_{j=1}^{p} |x_j - x'_j|^q \right)^{\frac{1}{q}}$$

For text or categorical data, metrics like **Hamming distance** or **Jaccard similarity** are used.

Tip: ***Importance of Scaling***

Since kNN calculates distances based on feature magnitudes, features with large ranges (e.g., Salary) will dominate features with small ranges (e.g., Age). **Standardization** (z-score) or Normalization of inputs is mandatory for kNN to work correctly.

The Hyperparameter k

The number of neighbors k is a **flexibility parameter** that controls the Bias-Variance trade-off.

- **Small** k (e.g., $k = 1$): The model is extremely flexible. The decision boundary is jagged and adapts to every single training point.
  - **Low Bias**: It fits the training data perfectly.
  - **High Variance**: It is very sensitive to noise (a single mislabeled neighbor flips the prediction).
- **Large** k (e.g., $k = n$): The model is very rigid. The decision boundary smooths out.
  - **High Bias**: It ignores local structure (at $k = n$, it just predicts the majority class of the dataset).
  - **Low Variance**: It is robust to noise in individual points.

Properties of kNN

- **Efficiency**: kNN struggles with large datasets in prediction. Each prediction requires computing distances to all n training points ($O(n \cdot p)$ complexity, where p is the number of features). There is no actual learning phase, the algorithm only stores the data.
- **Explainability/Interpretability**: The neighborhood itself is a local explanation of the decision. You can inspect the k nearest neighbors to understand why a prediction was made.
- **Effectiveness**: Not particularly good in practice. Performance depends on k: too small overfits, too large underfits.
- **Applicability**: Works for both regression and classification (Y). Handles any input space (X) if you have a proper distance metric d (though mixed numerical/categorical cases are tricky). Naturally gives probability estimates. Has two parameters (d and k), with k impacting bias-variance trade-off.

3 Unsupervised Learning

3.1 Introduction

In supervised learning, we are given examples (x, y) and the goal is to learn a mapping to predict y from x. In **unsupervised learning**, there are no labels y. We only observe the data X and we want to find interesting structures within it.

Definition: *Unsupervised Learning*

Unsupervised Learning is a type of machine learning that looks for previously undetected patterns in a dataset with no pre-existing labels and with a minimum of human supervision.

Since we do not have a "ground truth" to compare against, the goal is often to **discover** knowledge rather than to learn a specific predictor.

- The input is a dataset $D \in \mathcal{P}^*(X)$.
- The output is a **pattern** or structure describing the data.

3.2 Clustering

The most common form of unsupervised learning is **clustering**. We assume that the system generated the data following some grouping scheme, but we do not know the groups.

Definition: *Clustering*

Clustering is the task of grouping a set of objects in such a way that objects in the same group (called a ***cluster***) are more similar to each other than to those in other groups.

Formally, given a dataset $D \in \mathcal{P}^*(X)$, we want to find a partitioning $\mathcal{C} = \{D_1, \ldots, D_k\}$ such that:

- $\bigcup_{i=1}^{k} D_i = D$ (the union covers all data);
- $D_i \cap D_j = \emptyset$ for $i \neq j$ (clusters do not overlap);
- Elements within D_i are "close" to each other;
- Elements in different clusters D_i, D_j are "far" from each other.

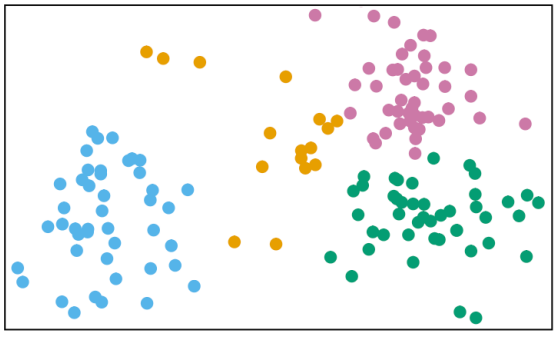


Figure 3.1: Visual representation of clustering.

Clustering as Optimization

Clustering can be viewed as an optimization problem where we try to maximize distinction between groups. We want to maximize **inter-cluster distance** (separation) and minimize **intra-cluster distance** (compactness). Let $d : X \times X \to \mathbb{R}^+$ be a distance metric. The objective function is:

$$\arg\max_{D_1,\dots,D_k} \left(\left(\sum_{i \neq j} \sum_{x \in D_i, x' \in D_j} d(x,x') \right) - \left(\sum_{i} \sum_{x,x' \in D_i} d(x,x') \right) \right)$$

subject to $\bigcup_{i=1}^{k} D_i = D$ and $D_i \cap D_j = \emptyset$ for $i \neq j$.
However, finding the global optimum is often computationally unfeasible. Furthermore, the number of clusters k cannot simply be optimized without constraints, because trivial solutions exist:

- If $k = 1$, compactness is minimized (bad), but separation is undefined.
- If $k = n$ (every point is a cluster), compactness is perfect (0), but separation is uninformative.

Therefore, k is usually a hyperparameter or determined by heuristics.

Assessing Clustering

Since we often lack labels, assessment is difficult.

1. **Manual Inspection**: Visualize the data (e.g., using PCA) and verify if groups make sense.
2. **Extrinsic Evaluation**: Use clusters in a downstream task and check if performance improves.
3. **Intrinsic Evaluation**: Measure mathematical properties of clusters (compactness, separation).

The Silhouette Index

The **Silhouette Index** measures how similar an object is to its own cluster compared to others.

Definition: *Silhouette Score*

For a single data point $x \in D_i$ (where D_i is the cluster x belongs to):

1. **mean intra-cluster distance** $a(x)$:
   average distance between x and all other points in D_i.

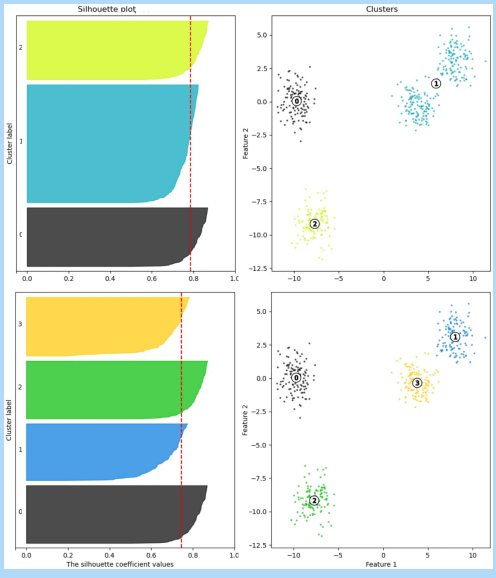
$$a(x) = \frac{1}{|D_i| - 1} \sum_{x' \in D_i, x' \neq x} d(x,x')$$

2. **mean nearest-cluster distance** $b(x)$:
   average distance between x and all points in the nearest neighboring cluster.

$$b(x) = \min_{j \neq i} \frac{1}{|D_j|} \sum_{x' \in D_j} d(x,x')$$

3. The **silhouette** $s(x)$ is:

$$s(x) = \frac{b(x) - a(x)}{\max\{a(x), b(x)\}}$$



The global Silhouette Score is the mean of $s(x)$ over all observations: $S = \frac{1}{n} \sum_x s(x)$.

Interpretation of $s(x) \in [-1, 1]$:

- **Close to 1**: The point is well-clustered (far from neighbors, close to its own group).
- **Close to 0**: The point is on the boundary between two clusters.
- **Close to** -1: The point is likely assigned to the wrong cluster.

3.3 Hierarchical Clustering

Hierarchical clustering builds a clusters' hierarchy, often represented as a tree (or **dendrogram**). It doesn't require specifying k in advance; cutting the tree at different depths yields different partitions.

Definition: ***Hierarchical Clustering Approaches***

- **Agglomerative (Bottom-Up)**: Start with n clusters (each point is a cluster). At each step, merge the two closest clusters until only one remains.
- **Divisive (Top-Down)**: Start with 1 cluster (containing all points). At each step, split a cluster until n clusters remain.

3.3.1 Agglomerative Clustering Algorithm

Algorithm 4 Agglomerative Hierarchical Clustering

Input: Dataset $X = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$, distance metric $d(\cdot,\cdot)$, linkage criterion
Output: Dendrogram (hierarchy of clusters)
Initialize: Assign each observation to its own cluster: $\mathcal{C} = \{\{x^{(1)}\}, \{x^{(2)}\}, \ldots, \{x^{(n)}\}\}$
Compute distance matrix: Calculate pairwise distances $D_{ij} = d(C_i, C_j)$ for all cluster pairs
while $|\mathcal{C}| > 1$ **do**
    Find the pair of clusters (C_i, C_j) with minimum distance: $(i^*, j^*) = \arg\min_{i<j} D_{ij}$
    Merge: Create new cluster $C_{i^*} \cup C_{j^*}$
    Update: Remove C_{i^*} and C_{j^*} from $\mathcal{C}$, add C_{new}
    Update distance matrix: Compute distances from C_{new} to all remaining clusters
    Record merge: Store the merge in the dendrogram structure
end while
return the dendrogram representing the complete hierarchy

Complexity: $O(n^3)$ for naive implementation; $O(n^2 \log n)$ with efficient data structures.

Linkage Methods (Cluster Distance)

The distance between two clusters D_i and D_j is defined by the **linkage criterion**, which strongly affects the shape of the resulting clusters.

- **Single Linkage**: Distance between the **closest** pair of points.

$$d(D_i, D_j) = \min_{x \in D_i, x' \in D_j} d(x, x')$$

  Effect: Can produce long, "chain-like" clusters.

- **Complete Linkage**: Distance between the **farthest** pair of points.

$$d(D_i, D_j) = \max_{x \in D_i, x' \in D_j} d(x, x')$$

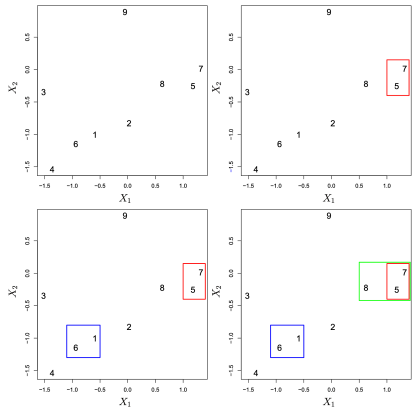
  Effect: Tends to produce compact, spherical clusters.

- **Average Linkage**: Average distance between all pairs.

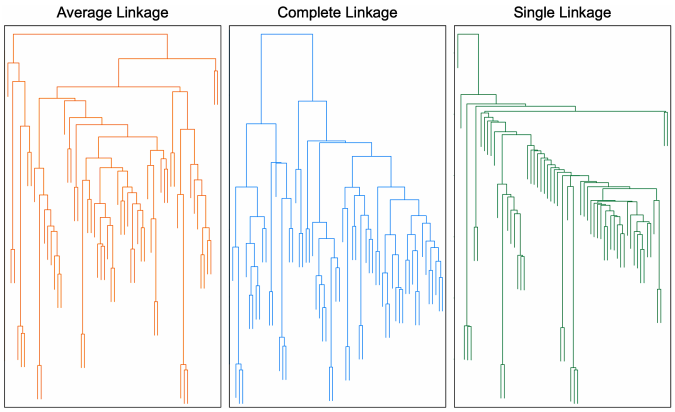
$$d(D_i, D_j) = \frac{1}{|D_i| \cdot |D_j|} \sum_{x \in D_i} \sum_{x' \in D_j} d(x, x')$$

- **Centroid Linkage**: Distance between the centroids (means) of the clusters.

$$d(D_i, D_j) = d(\mu_i, \mu_j)$$



(a) Dendrogram of hierarchical clustering.



(b) Different linkage criteria.

Figure 3.2: Linkage methods and dendrogram representation.

3.4 k-Means Clustering

k-Means is a **partitional** clustering algorithm. Unlike hierarchical methods, it requires the number of clusters k to be specified in advance. It attempts to separate samples in k groups of equal variance, minimizing a criterion known as **inertia** or within-cluster sum-of-squares (WCSS).

Definition: ***k-Means Algorithm***

Given a set of observations $(x^{(1)}, x^{(2)}, \ldots, x^{(n)})$, partition the n observations into k sets $S = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares. The algorithm is as follows:

1. **Initialization**: Choose k initial centroids $\{\mu_1, \ldots, \mu_k\}$ (randomly or using strategies as k-means++).
2. **Assignment**: Assign each observation to the cluster with the nearest mean.

$$S_i^{(t)} = \{x^{(j)} : \|x^{(j)} - \mu_i^{(t)}\|^2 \leq \|x^{(j)} - \mu_l^{(t)}\|^2 \; \forall l\}$$

3. **Update**: Recalculate centroids as the mean of observations assigned to the cluster.

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x^{(j)} \in S_i^{(t)}} x^{(j)}$$

4. **Repeat** steps 2 and 3 until convergence (centroids do not change).

- **Pros:** Efficient ($O(t \cdot k \cdot n)$ per iteration), simple to implement.
- **Cons:** Requires to set k; sensitive to initialization (local minima) and outliers; assumes spherical clusters.

Tip: ***Sensitivity to Initialization***

k-Means is not deterministic, due to the random initialization of the centroids. It is sensitive to the initial centroids, so it is better to run it multiple times and choose the result with the lowest inertia.

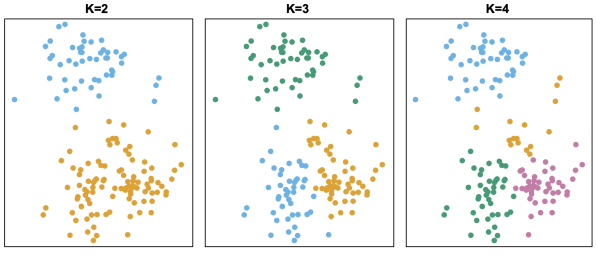


Figure 3.3: k-Means iterations.

4 Introduction to Natural Language Processing

4.1 Applying Machine Learning to Text

Text data differs significantly from the numerical or categorical data we have seen so far. It is unstructured, variable in length, and inherently high-dimensional.

Definition: *Text*

A ***text*** is a sequence of symbols (characters, words, or sentences) belonging to an alphabet A. Formally:

$$x \in A^*$$

where A^* is the set of all possible strings of symbols from A (in modern contexts, usually UTF-16).

In NLP terminology:

- A single text x is called a **document**.
- A dataset D of texts is called a **corpus**.

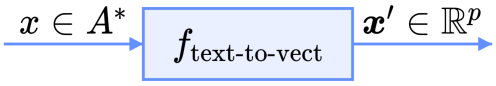


Figure 4.1: The general NLP pipeline.

Example: *Sentiment Analysis*

A classic application is **Sentiment Analysis**. The goal is to gain insights about the sentiments an author was feeling while writing a document. This is a standard **supervised learning problem**:

- Input X: A set of documents (strings).
- Output Y: A sentiment label (e.g., $Y = \{\text{Positive}, \text{Negative}, \text{Neutral}\}$).

Since standard ML models (like SVMs or Random Forests) require numerical vectors as input, we must define a **text-to-vector** transformation $f_{\text{text-to-vect}} : A^* \to \mathbb{R}^p$ to convert raw text into multivariate observations.

4.1.1 Feature Engineering: From Text to Vectors

The process of converting text into vectors involves two main stages: **pre-processing** (cleaning and normalizing the text) and **vectorization** (mapping tokens to numbers).

Text Pre-processing

Raw text often contains variations that are irrelevant for the learning task but increase the complexity of the model. Pre-processing functions $f_{\text{pre-proc}} : A^* \to A^*$ aim to **normalize** the text. Common steps include:

- **Case conversion**: Converting everything to lowercase to ensure "Apple" and "apple" are treated as the same token (language independent).
- **Removal of punctuation**: Stripping non-alphanumeric characters (language independent).
- **Removal of stop-words**: Removing very common words (e.g., articles, prepositions like "the", "is", "at") that carry little semantic meaning but appear frequently (language dependent).
- **Stemming**: Replacing words with their morphological root to group similar concepts (e.g., "liked", "likes" $\to$ "lik"). This reduces the vocabulary size significantly (language dependent).

Bag of Words (BoW)

The **Bag-of-Words** (BoW) model is a vectorization technique based on the idea that the *content* of a document is determined by the words it contains, ignoring grammar and order. It relies on a predefined **dictionary** (or vocabulary) $W = \{w_1, \ldots, w_p\}$, which is a set of unique words found in the corpus.

Definition: ***Bag of Words Representation***

Given a dictionary W of size p, the BoW function $f_{\text{BOW}} : A^* \to \mathbb{R}^p$ maps a document x to a vector x' where each component x'_j represents the **multiplicity** of the word w_j in x.

Formally, the process is:

1. **Tokenize** the document x into a multiset of tokens T.
2. For each word $w_j \in W$, count its occurrences in T.
3. The resulting vector is $x' = [c_1, c_2, \ldots, c_p]$, where c_j is the count of w_j.

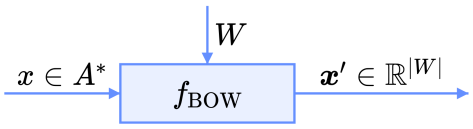


Figure 4.2: BoW vectorization.

An alternative to raw counts is using **frequencies**: $x'_j = \dfrac{\text{count}(w_j)}{|T|}$.

This normalizes the vector, making the representation robust to differences in document length.

TF-IDF

A major limitation of simple BoW counting is that it highlights words that are very frequent in the corpus but not necessarily informative (e.g., the word "good" might appear in almost every review, making it useless for distinguishing positive from negative ones). To address this, we use **TF-IDF** (Term Frequency-Inverse Document Frequency). It re-weights the counts to highlight words that are distinct to a specific document.

$$\text{tfidf}(t,d,D) = \text{tf}(t,d) \cdot \text{idf}(t,D)$$

Where:

- t is the term (word);
- d is the specific document;
- D is the corpus (collection of all documents);
- $\text{tf}(t,d)$ is the **Term Frequency**: frequency of t in d.
- $\text{idf}(t,D)$ is the **Inverse Document Frequency**:

$$\text{idf}(t,D) = \log\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right)$$

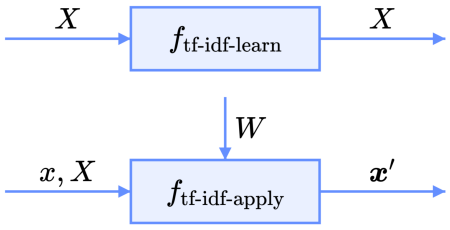


Figure 4.3: TF-IDF logic.

The IDF term measures how rare the term is across the corpus. If a word appears in many documents, the ratio inside the log approaches 1, and the IDF approaches 0. Intuitively, a high TF-IDF score indicates that a word is frequent in this document but rare in the corpus, meaning it characterizes this document.

Dimensionality and Context

With BoW methods, the dimension of the input space can be huge. Common reduction practices include:

- Using a very small, domain-specific dictionary.
- Pruning the dictionary based on frequency (e.g., keeping only the top-k most frequent words).
- Using dimensionality reduction techniques (e.g., PCA/SVD) on the resulting matrix.

The problem of Ordering: BoW destroys word order ("not bad" becomes "bad" + "not", losing the negation context). To capture context, we can use:

- **N-grams**: Instead of counting single words (1-grams), we count sequences of n adjacent words (e.g., 2-grams like "not bad", "very good"). This captures local context but drastically increases dimensionality.
- **Part of Speech (POS) Tagging**: Analyzing the grammatical role (noun, verb, adjective) of words to disambiguate meaning (e.g., "book" as a noun vs. "book" as a verb).