

Statistical Supervised Machine Learning

Master MIASHS & MALIA

Guillaume Metzler

Institut de Communication (ICOM)

Université de Lyon, Université Lumière Lyon 2

Laboratoire ERIC UR 3083, Lyon, France

guillaume.metzler@univ-lyon2.fr

Abstract

These lecture notes introduce supervised machine learning, from its mathematical foundations to the algorithms that are used in practice and to the theory that explains why they work. *Part I* gathers the mathematical background needed throughout the course: norms, derivatives, convex sets and convex functions, and convex optimization, with the optimality conditions, the gradient descent algorithms and the Lagrangian duality that underlie most learning methods.

Part II sets up the supervised learning problem. It describes the data and the main learning tasks, introduces the framework of statistical learning theory — empirical risk minimization and generalization bounds — and details the learning procedure: how to train a model, how to tune its hyper-parameters and how to evaluate it with performance measures adapted to regression and classification. It then presents the main supervised algorithms: surrogate losses, k -nearest neighbours, support vector machines, linear and logistic regression, multinomial logistic regression, kernel regression, decision trees and neural networks. For each of them, we describe the model, the optimization problem it solves and the way it is used in practice.

Part III goes one step further. It comes back to statistical learning theory through the decomposition of the error into bias, variance, approximation and estimation terms, and uses it to explain why combining models works: bagging and random forests, boosting, stacking and gradient boosting. Metric learning and practical advice on learning with real data complete this part.

Part IV presents PAC-Bayesian theory, a framework whose object of study is a distribution over models, that is, precisely what an ensemble method learns. It derives generalization bounds that certify the risk of majority votes, explains the role of the diversity of the voters, and shows how minimizing these bounds leads to *self-bounding* algorithms, which output a model together with a guarantee on its risk.

Each notion is illustrated with examples and figures, and each section ends with short questions to check one's understanding. The exercises of the course are gathered in a separate booklet whose sections follow those of these notes. For neural networks, we warmly recommend the video lectures of [Hugo Larochelle](#), an excellent complement to the corresponding section.

Contents

How to read these notes	6
I Mathematical background	7
1 Probability Essentials and Concentration Inequalities	8
1.1 Probability spaces, random variables and distributions	8
1.2 Expectation, variance and conditional expectation	12
1.3 Limit theorems: law of large numbers and central limit theorem	18
1.4 Concentration inequalities	21
2 Linear Algebra and Convexity	33
2.1 Norms	33
2.2 Derivatives	37
2.3 Convex sets	40
2.4 Convex functions	42
3 Unconstrained and Smooth Convex Optimization	52
3.1 Motivation	52
3.2 Optimality Conditions	53
3.3 Minimization Problems	57
3.4 Gradient Descent Algorithms	59
4 Constrained and Smooth Convex Optimization	73
4.1 Motivation: Constraints in Machine Learning	73
4.2 Problem Formulation	74
4.3 Equality Constraints and Lagrange Multipliers	75
4.4 Lagrangian Duality	77
4.5 The Karush–Kuhn–Tucker Conditions	79
4.6 Worked Examples	81
4.7 Projected Gradient Descent	82
II Supervised Machine Learning	87
5 Introduction	88
5.1 Generalities	88
5.2 Data and Machine Learning problems	89

6	Statistical Learning Theory	98
6.1	Empirical Risk Minimization	99
6.2	Generalization Bounds	104
7	Learning Procedure	113
7.1	How to train and tune a Model	113
7.2	Performance Measures	119
8	Supervised Algorithms	130
8.1	Surrogate losses	130
8.2	k -Nearest Neighbors	131
8.3	Support Vector Machine	137
8.4	Linear Regression and Logistic Regression	149
8.5	Multinomial Logistic Regression	156
8.6	Kernel Regression	159
8.7	Decision Trees (classification and regression)	161
9	Neural Networks	168
9.1	From the perceptron to the multilayer perceptron	168
9.2	Activation functions	174
9.3	Expressive power: the universal approximation theorem	176
9.4	Loss functions	177
9.5	Training: forward pass and backpropagation	179
9.6	Optimization	184
9.7	Regularization	186
9.8	Convolutional neural networks	187
9.9	Recurrent neural networks	191
9.10	Other architectures	192
9.11	Practical advice	195
III	Advanced Supervised Machine Learning and Implementation	198
10	Advanced Supervised Algorithms	199
10.1	Back to Statistical Learning Theory	199
10.2	Ensemble Methods	207
10.3	Gradient Boosting	227
10.4	Metric Learning	234

11 Applications and Learning in Practice	244
11.1 Framing the Problem and Choosing a Metric	244
11.2 Collecting and Exploring the Data	245
11.3 Preprocessing and Feature Engineering	245
11.4 Splitting the Data and Avoiding Leakage	247
11.5 Choosing, Tuning and Comparing Models	248
11.6 Learning from Imbalanced Data	250
11.7 Interpreting a Model	253
11.8 Reproducibility and Reporting	254
11.9 A Checklist	255
 IV PAC-Bayesian Learning: a Theoretical Framework for Ensemble Methods	 258
 12 Why PAC-Bayes for ensemble methods?	 261
12.1 A puzzling experiment	261
12.2 A vote is a distribution over voters	262
12.3 What is a guarantee?	264
12.4 The PAC-Bayesian idea	265
 13 From an error on a sample to a guarantee	 267
13.1 One classifier fixed in advance	267
13.2 Markov's inequality	267
13.3 Hoeffding's inequality and the test-set bound	268
13.4 A sharper bound with the binary kl	269
13.5 What goes wrong when we choose	272
13.6 Occam's bound: spreading the confidence with a prior	273
 14 The PAC-Bayesian bound	 276
14.1 Measuring the distance to the prior: the KL divergence	276
14.2 The change of measure	277
14.3 A first PAC-Bayesian bound	278
14.4 The bounds used in practice	279
14.5 Reading the bound	281
14.6 Choosing the prior and the posterior	282

15 From the Gibbs classifier to the majority vote	285
15.1 One random variable that describes the vote	285
15.2 The first-order bound	286
15.3 Condorcet's jury theorem	287
15.4 Pairs of voters: joint error and disagreement	287
15.5 The tandem bound	289
15.6 Margins and boosting	290
16 Learning with the bounds	293
16.1 From empirical risk minimization to bound minimization	293
16.2 Random forests and the out-of-bag examples	294
16.3 Learning the weights of a forest	294
16.4 Beyond ensembles	296
17 Deriving a self-bounding algorithm, step by step	298
17.1 The recipe	298
17.2 Steps 1 and 2: the bound and what is fixed	299
17.3 Step 3: the objective	299
17.4 Step 4: solving the minimization	300
17.5 Step 5: certify, and choose λ	302
17.6 What to remember from the derivation	303
A Answers to the “Check your understanding” questions	305
Bibliography	318

How to read these notes

The notes are organized in four parts, which can be read in order. **Part I** is a toolbox: readers familiar with convex optimization can skip it and come back to it when a result is needed. **Part II** is the core of the course (Master 1), **Part III** is devoted to ensemble methods and to the practice of machine learning, and **Part IV** develops the PAC-Bayesian theory of ensemble methods (Master 2 MALIA), in six short sections that start from the fundamentals.

Definitions are given in blue boxes, and theorems, propositions and corollaries in red boxes, always with their proofs or proof sketches; lemmas are in green boxes. Long computations are written one step per line, and a grey arrow $\downarrow$ like this explains how each line is obtained from the previous one. Examples are numbered by section.

Each section ends with a summary *To remember* and with short questions *Check your understanding*, grouped by subsection, whose answers are in **Section A**. The figures of **Part IV** are generated by Python scripts gathered in the folder `code/` of the project, or by TikZ files of the folder `figures/tikz/`; the end of each caption indicates its source.

Prerequisites. Basic linear algebra (vectors, matrices, inner products), differential calculus in several variables, and probability (expectation, variance, independence, the law of large numbers).

Part I

Mathematical background

Machine learning algorithms are, for the most part, solutions of optimization problems: a model is learned by minimizing a criterion that measures how badly it fits the data, often with a penalty on its complexity. This first part recalls the mathematical tools needed to write, understand and solve such problems. Since models are learned from random data, [Section 1](#) first recalls the probability needed to describe this randomness: expectations, conditional expectations, limit theorems and the concentration inequalities that turn an error measured on a sample into a guarantee on unseen data. [Section 2](#) gathers the notions of linear algebra and convexity (norms, derivatives, convex sets and convex functions); [Section 3](#) studies unconstrained smooth convex optimization, its optimality conditions and the gradient descent algorithms that underlie most learning methods; [Section 4](#) deals with constrained problems, Lagrangian duality and the Karush–Kuhn–Tucker conditions, which are the key to support vector machines. These reminders are essential to understand the calculations and properties of the algorithms presented later, and the way solvers work. For further details on convexity and convex optimization, we refer to [Boyd and Vandenberghe \(2004\)](#).

1 Probability Essentials and Concentration Inequalities

Machine learning is about learning from *data*, and data are random: the examples that we observe could have been different, and the examples on which our model will be used are not known yet. Probability theory is the language that allows us to describe this randomness, to define precisely what a “good” model is, and, above all, to quantify how much we can trust what we measure on a finite sample. This first section gathers the notions of probability used throughout the notes. It starts from scratch, with intuition and examples (a very readable complement is the book of Wasserman (2004)), so that it can be read by students who have not practised probability for a while, but it remains rigorous, since the results stated here are used in the proofs of the following sections.

About this section

Guiding question. *A classifier makes 12% of errors on 1000 examples: what does this tell us about the errors it will make on new examples, and how confident can we be?*

In this section you will learn to:

- describe a random experiment with a probability space, random variables and their distributions (probability mass functions, densities, cumulative distribution functions), and use the i.i.d. assumption that underlies supervised learning;
- compute expectations, variances and conditional expectations, and recognize that the risk of a predictor is an expectation and the empirical risk an average;
- state the law of large numbers and the central limit theorem, and explain why they are not sufficient to certify a learning algorithm;
- prove and use the union bound and the inequalities of Markov, Chebyshev, Chernoff and Hoeffding, use those of Bernstein and McDiarmid, and turn them into statements “with probability at least $1 - \delta$ ” and into sample sizes.

In a nutshell The true performance of a model is an *expectation* over the unknown distribution of the data; what we can compute is an *average* over a finite sample. The law of large numbers says that the average converges to the expectation; *concentration inequalities* say how far apart they can be for a given sample size m , with a given confidence. Typically, the deviation is of order $\sqrt{\ln(1/\delta)/m}$: to halve the uncertainty, one needs four times more data, whereas a much higher confidence costs very little.

Throughout the section, we use the following running question: a classifier h has been fixed (for instance, it was trained on other data), and we evaluate it on m new examples on which it makes $\hat{p}$ errors in proportion. What can we say about its true error rate p ? We will meet this question again and again, with sharper and sharper answers.

1.1 Probability spaces, random variables and distributions

Random experiments

A *random experiment* is an experiment whose outcome cannot be predicted with certainty: rolling a die, drawing a customer at random in a database, observing tomorrow’s

temperature, or drawing a training sample. Its mathematical description is a probability space.

Definition 1.1: Probability space

A **probability space** is a triple $(\Omega, \mathcal{F}, \mathbb{P})$ where

- Ω is the set of all possible outcomes ω of the experiment (the *sample space*);
- $\mathcal{F}$ is a collection of subsets of Ω , called *events*, which contains Ω and is stable under complement and countable unions (a σ -algebra);
- $\mathbb{P} : \mathcal{F} \rightarrow [0, 1]$ is a *probability measure*: $\mathbb{P}(\Omega) = 1$ and, for every sequence $(A_k)_{k \geq 1}$ of pairwise disjoint events,

$$\mathbb{P}\left(\bigcup_{k \geq 1} A_k\right) = \sum_{k \geq 1} \mathbb{P}(A_k) \quad (\sigma\text{-additivity}).$$

The σ -algebra is a technical ingredient that we will not discuss further: in this course, every set that we write down is an event. The axioms immediately give the rules that you already know: $\mathbb{P}(\emptyset) = 0$, $\mathbb{P}(A^c) = 1 - \mathbb{P}(A)$, $\mathbb{P}(A) \leq \mathbb{P}(B)$ if $A \subseteq B$, and $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B)$.

Example 1.1. For a fair die, $\Omega = \{1, \dots, 6\}$, every subset is an event and $\mathbb{P}(A) = |A|/6$. The event “the result is even” is $A = \{2, 4, 6\}$ and $\mathbb{P}(A) = 1/2$. For two independent tosses of a coin with probability p of heads, $\Omega = \{HH, HT, TH, TT\}$ and $\mathbb{P}(\{HT\}) = p(1 - p)$.

The *conditional probability* of A given an event B with $\mathbb{P}(B) > 0$ is $\mathbb{P}(A | B) = \mathbb{P}(A \cap B) / \mathbb{P}(B)$: it is the proportion of the outcomes of B that also belong to A . If $(B_k)_k$ is a partition of Ω into events of positive probability, we get the *law of total probability* and *Bayes’ formula*:

$$\mathbb{P}(A) = \sum_k \mathbb{P}(A | B_k) \mathbb{P}(B_k), \quad \mathbb{P}(B_j | A) = \frac{\mathbb{P}(A | B_j) \mathbb{P}(B_j)}{\sum_k \mathbb{P}(A | B_k) \mathbb{P}(B_k)}. \quad (1.1)$$

Random variables and their distributions

In practice, we are not interested in the outcome ω itself, but in numbers computed from it: the size of a customer, the label of an image, the number of errors of a classifier.

Definition 1.2: Random variable, distribution, cumulative distribution function

A (real) **random variable** is a function $X : \Omega \rightarrow \mathbb{R}$ such that $\{X \leq t\} = \{\omega \in \Omega : X(\omega) \leq t\}$ is an event for every $t \in \mathbb{R}$. The **distribution** (or *law*) of X is the probability $\mathbb{P}_X$ on $\mathbb{R}$ defined by $\mathbb{P}_X(B) = \mathbb{P}(X \in B)$, and its **cumulative distribution function** (cdf) is

$$F_X(t) = \mathbb{P}(X \leq t), \quad t \in \mathbb{R}.$$

A cdf is non-decreasing, right-continuous, tends to 0 at $-\infty$ and to 1 at $+\infty$, and it characterizes the distribution: two random variables with the same cdf have the same distribution, which we write $X \sim Y$ (or $X \sim \mathbb{P}_X$). Two kinds of distributions will cover all our needs.

- X is **discrete** if it takes its values in a finite or countable set $\{x_1, x_2, \dots\}$. Its distribution is described by its *probability mass function* (pmf) $p_X(x_k) = \mathbb{P}(X = x_k)$, with $\sum_k p_X(x_k) = 1$, and $\mathbb{P}(X \in B) = \sum_{k: x_k \in B} p_X(x_k)$.
- X is **continuous** (or *has a density*) if there exists a nonnegative function f_X with $\int_{\mathbb{R}} f_X(x) dx = 1$ such that $\mathbb{P}(X \in [a, b]) = \int_a^b f_X(x) dx$ for all $a \leq b$. Then $F_X(t) = \int_{-\infty}^t f_X(x) dx$ and $f_X = F'_X$ wherever f_X is continuous.

Pitfall A density is *not* a probability: $f_X(x)$ can be larger than 1 (see the density of $\mathcal{N}(1, 0.25)$ in Fig. 1), and for a continuous variable $\mathbb{P}(X = x) = 0$ for every x . Only areas under the density are probabilities.

The following distributions will appear everywhere in this course.

- **Bernoulli** $\mathcal{B}(p)$, $p \in [0, 1]$: $X \in \{0, 1\}$ with $\mathbb{P}(X = 1) = p$. It models any yes/no outcome; in particular, the indicator $\mathbb{1}_{\{h(\mathbf{x}) \neq y\}}$ that a classifier h errs on a random example $(\mathbf{x}, y)$ is a Bernoulli variable whose parameter is the error rate of h .
- **Binomial** $\text{Bin}(m, p)$: the number of successes in m independent Bernoulli trials of parameter p , with $\mathbb{P}(X = k) = \binom{m}{k} p^k (1-p)^{m-k}$ for $k \in \{0, \dots, m\}$. The number of errors of a fixed classifier on m independent test examples is binomial.
- **Uniform** $\mathcal{U}([a, b])$, $a < b$: density $f(x) = \frac{1}{b-a}$ on $[a, b]$ and 0 elsewhere. All the sub-intervals of $[a, b]$ of the same length have the same probability.
- **Gaussian** (or normal) $\mathcal{N}(\mu, \sigma^2)$, $\mu \in \mathbb{R}$, $\sigma > 0$: density $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2})$. If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $(X - \mu)/\sigma \sim \mathcal{N}(0, 1)$, whose cdf is denoted by Φ .

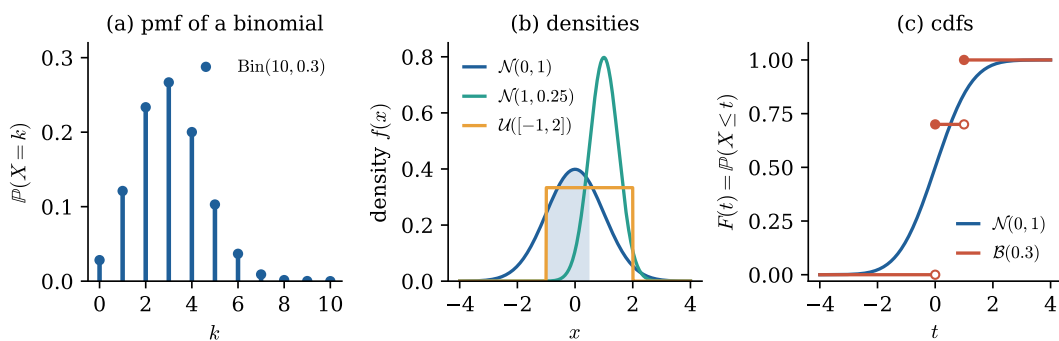


Figure 1: (a) The pmf of the binomial distribution $\text{Bin}(10, 0.3)$, e.g. the number of errors on 10 test examples of a classifier with error rate 0.3. (b) Densities of two Gaussian distributions and of a uniform distribution; the shaded area is $\mathbb{P}(-1 \leq X \leq 0.5)$ for $X \sim \mathcal{N}(0, 1)$, and a density can exceed 1. (c) The cdf of $\mathcal{N}(0, 1)$ is continuous, while the cdf of the Bernoulli distribution $\mathcal{B}(0.3)$ is a step function, which jumps by $\mathbb{P}(X = x)$ at each value x (filled dots: value of the cdf at the jump).

[code: make_figures_prob.py: fig_prob_distributions]

Random vectors, joint distributions and independence

A **random vector** $\mathbf{X} = (X_1, \dots, X_d)$ is a vector of random variables defined on the same probability space; its distribution on $\mathbb{R}^d$ is called the *joint distribution* of $X_1, \dots, X_d$. In supervised learning, the pair $(\mathbf{x}, y)$ made of a description $\mathbf{x} \in \mathbb{R}^d$ and of a label y is a random vector, whose joint distribution is the (unknown) distribution $\mathcal{D}$ of the data.

The distribution of a single component is called a *marginal distribution*; it is obtained by summing (or integrating) the joint pmf (or density) over the other components. For instance, for two discrete variables,

$$\mathbb{P}(X = x) = \sum_y \mathbb{P}(X = x, Y = y), \quad \text{and, with densities,} \quad f_X(x) = \int_{\mathbb{R}} f_{X,Y}(x, y) dy.$$

The marginals do *not* determine the joint distribution: knowing the distribution of the heights and the distribution of the weights of a population does not tell us how heights and weights are related. Independence is the special case where there is no relation at all.

Definition 1.3: Independence

Random variables $X_1, \dots, X_n$ are **(mutually) independent** if, for all sets $B_1, \dots, B_n$,

$$\mathbb{P}(X_1 \in B_1, \dots, X_n \in B_n) = \prod_{i=1}^n \mathbb{P}(X_i \in B_i).$$

Equivalently, the joint pmf (or the joint density) is the product of the marginal ones. They are **independent and identically distributed** (*i.i.d.*) if, moreover, they all have the same distribution.

If $X_1, \dots, X_n$ are independent, then so are $g_1(X_1), \dots, g_n(X_n)$ for any functions $g_1, \dots, g_n$, and so are functions of disjoint groups of variables, such as $X_1 + X_2$ and $X_3 X_4$.

Pitfall Independence of all pairs does not imply mutual independence. Let X_1, X_2 be independent and uniform on $\{-1, +1\}$, and $X_3 = X_1 X_2$. Each pair is independent, but X_3 is a function of (X_1, X_2) : knowing X_1 and X_2 , we know X_3 .

The i.i.d. assumption in machine learning

Throughout the notes, we assume that the examples of the learning sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ are drawn *independently* from the *same* distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, and that the future examples on which the model will be used are drawn from $\mathcal{D}$ as well. We write $S \sim \mathcal{D}^m$, where $\mathcal{D}^m$ denotes the product distribution: the sample S is itself a random object, and every quantity computed from it (the empirical risk, the learned model, its accuracy on a test set) is a random variable.

This assumption is what makes learning possible at all. “Identically distributed” means that the past is representative of the future: the examples we learn from look like the examples on which we will predict. “Independent” means that each new example

brings new information: the empirical risk then becomes an average of independent variables, and all the tools of this section apply to it.

Remark The i.i.d. assumption is a model, and it can fail in practice: time series (today's stock price depends on yesterday's), groups of correlated examples (several images of the same patient, several sales of the same customer), duplicates between training and test data, or a distribution that changes over time (*distribution shift*). In these cases, the guarantees of this course do not apply as such, and the evaluation protocol must be adapted, for instance by splitting the data by patient or by date (see Section 6 and the following sections).

1.2 Expectation, variance and conditional expectation

Expectation

The expectation of a random variable is its mean value, where each possible value is weighted by its probability. It is the “centre of mass” of its distribution.

Definition 1.4: Expectation

Let X be a random variable.

- If X is discrete with values $x_1, x_2, \dots$ and $\sum_k |x_k| \mathbb{P}(X = x_k) < +\infty$, its **expectation** is $\mathbb{E}[X] = \sum_k x_k \mathbb{P}(X = x_k)$.
- If X has a density f_X and $\int |x| f_X(x) dx < +\infty$, its expectation is $\mathbb{E}[X] = \int_{\mathbb{R}} x f_X(x) dx$.

More generally, for a function $g : \mathbb{R} \rightarrow \mathbb{R}$ (and in the same way for a random vector $\mathbf{X}$ and $g : \mathbb{R}^d \rightarrow \mathbb{R}$),

$$\mathbb{E}[g(X)] = \sum_k g(x_k) \mathbb{P}(X = x_k) \quad \text{or} \quad \mathbb{E}[g(X)] = \int_{\mathbb{R}} g(x) f_X(x) dx,$$

whenever the sum or the integral converges absolutely. The expectation of a random vector is the vector of the expectations of its components.

The last formula (sometimes called the *transfer theorem*) is very convenient: to compute $\mathbb{E}[X^2]$ or $\mathbb{E}[e^{\lambda X}]$, we do not need the distribution of X^2 or of $e^{\lambda X}$, only that of X . When the expectation is taken with respect to a variable whose distribution must be made explicit, we write it below the symbol, as in $\mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}}[\ell(h(\mathbf{x}), y)]$ or $\mathbb{E}_{S \sim \mathcal{D}^m}[\cdot]$.

Example 1.2. • If $X \sim \mathcal{B}(p)$, then $\mathbb{E}[X] = 0 \times (1 - p) + 1 \times p = p$.

- If $X \sim \mathcal{U}([a, b])$, then $\mathbb{E}[X] = \frac{1}{b-a} \int_a^b x dx = \frac{a+b}{2}$.
- If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $\mathbb{E}[X] = \mu$ (the density is symmetric around μ), and $\mathbb{E}[e^{\lambda X}] = e^{\lambda\mu + \lambda^2\sigma^2/2}$ for every $\lambda \in \mathbb{R}$ (complete the square in the exponent).
- For the indicator $\mathbb{1}_A$ of an event A (equal to 1 on A and 0 elsewhere), $\mathbb{E}[\mathbb{1}_A] = \mathbb{P}(A)$.

Proposition 1.1: Properties of the expectation

Let X, Y be random variables with finite expectations, and $a, b \in \mathbb{R}$.

- (i) **Linearity:** $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$, whether or not X and Y are independent.
- (ii) **Monotonicity:** if $X \geq 0$ almost surely, then $\mathbb{E}[X] \geq 0$; hence $X \leq Y$ implies $\mathbb{E}[X] \leq \mathbb{E}[Y]$. Moreover, $|\mathbb{E}[X]| \leq \mathbb{E}[|X|]$.
- (iii) **Indicators:** for every event A , $\mathbb{E}[\mathbb{1}_A] = \mathbb{P}(A)$.
- (iv) **Independence:** if X and Y are independent, then $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$; more generally, $\mathbb{E}[g(X)k(Y)] = \mathbb{E}[g(X)]\mathbb{E}[k(Y)]$ for all functions g, k such that these expectations are finite. In particular, for independent $X_1, \dots, X_n$ and $\lambda \in \mathbb{R}$, $\mathbb{E}[e^{\lambda \sum_i X_i}] = \prod_i \mathbb{E}[e^{\lambda X_i}]$.
- (v) **Tail formula:** if $X \geq 0$, then

$$\mathbb{E}[X] = \int_0^{+\infty} \mathbb{P}(X > t) dt,$$

and, if X takes values in $\mathbb{N}$, $\mathbb{E}[X] = \sum_{k \geq 1} \mathbb{P}(X \geq k)$.

Proof. (i)–(iii) follow from the linearity and the monotonicity of sums and integrals. For (iv) in the discrete case,

$$\begin{aligned} \mathbb{E}[XY] &= \sum_{x,y} xy \mathbb{P}(X = x, Y = y) = \sum_{x,y} xy \mathbb{P}(X = x) \mathbb{P}(Y = y) \\ &= \left(\sum_x x \mathbb{P}(X = x) \right) \left(\sum_y y \mathbb{P}(Y = y) \right), \end{aligned}$$

and the case of densities is identical, with integrals; the product formula for $e^{\lambda \sum_i X_i} = \prod_i e^{\lambda X_i}$ follows by induction on n . For (v), write $X = \int_0^X dt = \int_0^{+\infty} \mathbb{1}_{\{t < X\}} dt$ and exchange the expectation and the integral (this is allowed for nonnegative integrands, by Tonelli's theorem):

$$\mathbb{E}[X] = \mathbb{E} \left[\int_0^{+\infty} \mathbb{1}_{\{t < X\}} dt \right] = \int_0^{+\infty} \mathbb{E} [\mathbb{1}_{\{t < X\}}] dt = \int_0^{+\infty} \mathbb{P}(X > t) dt.$$

For an integer-valued variable, $\mathbb{P}(X > t) = \mathbb{P}(X \geq k)$ for $t \in [k-1, k)$, which gives the sum. $\square$

Remark Linearity holds *without any independence assumption*, and this is what makes it so useful: the expected number of errors of a classifier on m examples is m times its error rate, even if the examples were not independent. On the contrary, the product rule (iv) requires independence: if $X \sim \mathcal{B}(1/2)$, then $\mathbb{E}[X \cdot X] = 1/2 \neq 1/4 = \mathbb{E}[X]^2$.

Variance and covariance

The expectation tells us where a variable is on average; the variance tells us how much it fluctuates around this average.

Definition 1.5: Variance, covariance

Let X, Y be random variables with $\mathbb{E}[X^2] < +\infty$ and $\mathbb{E}[Y^2] < +\infty$. The **variance** of X and the **covariance** of X and Y are

$$\begin{aligned}\text{Var}(X) &= \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2, \\ \text{Cov}(X, Y) &= \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y].\end{aligned}$$

The **standard deviation** of X is $\sigma_X = \sqrt{\text{Var}(X)}$, which has the same unit as X .

Proposition 1.2: Properties of the variance

Let $X, X_1, \dots, X_m$ be random variables with finite second moments and $a, b \in \mathbb{R}$.

- (i) $\text{Var}(aX + b) = a^2 \text{Var}(X)$, and $\text{Var}(X) = 0$ if and only if X is almost surely constant.
- (ii) $\text{Var}\left(\sum_{i=1}^m X_i\right) = \sum_{i=1}^m \text{Var}(X_i) + 2 \sum_{i < j} \text{Cov}(X_i, X_j)$.
- (iii) If $X_1, \dots, X_m$ are independent (or only pairwise uncorrelated), the variance of the sum is the sum of the variances. In particular, if they are i.i.d. with variance σ^2 , the empirical mean $\bar{X}_m = \frac{1}{m} \sum_{i=1}^m X_i$ satisfies

$$\mathbb{E}[\bar{X}_m] = \mathbb{E}[X_1] \quad \text{and} \quad \text{Var}(\bar{X}_m) = \frac{\sigma^2}{m}.$$

- (iv) **Bounded variables:** if $a \leq X \leq b$ almost surely, then $\text{Var}(X) \leq \frac{(b-a)^2}{4}$.

Proof. (i) and (ii) follow from the definition by expanding the squares and using the linearity of the expectation. (iii) Independence implies $\text{Cov}(X_i, X_j) = 0$ by Proposition 1.1 (iv); then $\text{Var}(\bar{X}_m) = \frac{1}{m^2} \sum_i \text{Var}(X_i) = \frac{\sigma^2}{m}$. (iv) For every $c \in \mathbb{R}$, $\mathbb{E}[(X - c)^2] = \text{Var}(X) + (\mathbb{E}[X] - c)^2 \geq \text{Var}(X)$; with $c = \frac{a+b}{2}$, $|X - c| \leq \frac{b-a}{2}$, hence $\text{Var}(X) \leq \mathbb{E}[(X - c)^2] \leq \frac{(b-a)^2}{4}$. $\square$

The formula $\text{Var}(\bar{X}_m) = \sigma^2/m$ is the first quantitative statement about averages in this course: the standard deviation of an average of m independent measurements decreases as $\sigma/\sqrt{m}$. This $1/\sqrt{m}$ rate will appear in every generalization bound.

Example 1.3. If $X \sim \mathcal{B}(p)$, then $X^2 = X$, so that $\text{Var}(X) = p - p^2 = p(1 - p) \leq \frac{1}{4}$, with equality for $p = \frac{1}{2}$. If $X \sim \text{Bin}(m, p)$, it is a sum of m independent Bernoulli variables, hence $\mathbb{E}[X] = mp$ and $\text{Var}(X) = mp(1 - p)$. For $X \sim \mathcal{U}([a, b])$, $\text{Var}(X) = \frac{(b-a)^2}{12}$, and for $X \sim \mathcal{N}(\mu, \sigma^2)$, $\text{Var}(X) = \sigma^2$.

Pitfall Independent variables are uncorrelated, but the converse is false. If X is uniform on $\{-1, 0, 1\}$ and $Y = X^2$, then $\text{Cov}(X, Y) = \mathbb{E}[X^3] - \mathbb{E}[X]\mathbb{E}[X^2] = 0$, although Y is a function of X . The covariance only measures *linear* dependence.

Jensen's inequality

Expectation and nonlinear functions do not commute: in general $\mathbb{E}[g(X)] \neq g(\mathbb{E}[X])$. For convex functions, the two quantities are nevertheless always ordered. Recall (Definition 2.7 in Section 2) that f is convex if every chord of its graph lies above the graph.

Proposition 1.3: Jensen's inequality

Let X be a random variable with values in an interval I , with $\mathbb{E}|X| < +\infty$, and let $f : I \rightarrow \mathbb{R}$ be a convex function such that $\mathbb{E}|f(X)| < +\infty$. Then

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)].$$

If f is concave, the inequality is reversed: $\mathbb{E}[f(X)] \leq f(\mathbb{E}[X])$.

Proof. We give the proof for a differentiable convex function (the general case uses a supporting line instead of the tangent). Let $\mu = \mathbb{E}[X] \in I$. A differentiable convex function lies above its tangents (Proposition 2.6): $f(x) \geq f(\mu) + f'(\mu)(x - \mu)$ for every $x \in I$. Applying this with $x = X$ and taking the expectation, by monotonicity and linearity,

$$\mathbb{E}[f(X)] \geq f(\mu) + f'(\mu)(\mathbb{E}[X] - \mu) = f(\mu).$$

For a concave f , apply the result to the convex function $-f$. □

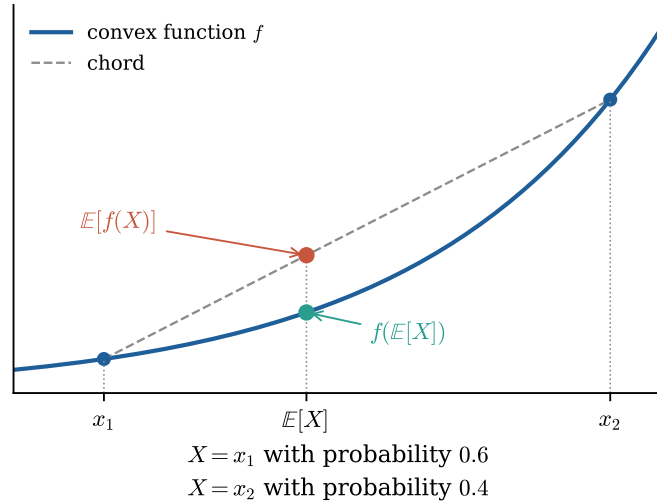


Figure 2: Jensen's inequality for a variable taking two values. $\mathbb{E}[f(X)]$ is a weighted average of $f(x_1)$ and $f(x_2)$, i.e. a point of the chord (red), while $f(\mathbb{E}[X])$ is the point of the graph (teal) at the same abscissa. For a convex f , the chord lies above the graph. [code:

make_figures_prob.py: fig_prob_jensen]

Jensen's inequality is used constantly in learning theory. With $f(x) = x^2$, it gives $\mathbb{E}[X]^2 \leq \mathbb{E}[X^2]$, i.e. $\text{Var}(X) \geq 0$. With the concave functions $\sqrt{\cdot}$ and $\ln$, it gives $\mathbb{E}[\sqrt{Z}] \leq \sqrt{\mathbb{E}[Z]}$ and $\mathbb{E}[\ln Z] \leq \ln \mathbb{E}[Z]$ for $Z > 0$; with the convex function $\exp$, it gives $e^{\mathbb{E}[X]} \leq \mathbb{E}[e^X]$. The last two inequalities are the heart of the change of measure inequality of PAC-Bayesian theory (see Section 1.4).

Pitfall $\mathbb{E}[1/X] \neq 1/\mathbb{E}[X]$ in general. For X uniform on $\{1, 3\}$, $\mathbb{E}[1/X] = \frac{2}{3}$ whereas $1/\mathbb{E}[X] = \frac{1}{2}$: by Jensen's inequality applied to the convex function $x \mapsto 1/x$ on $(0, +\infty)$, $\mathbb{E}[1/X] \geq 1/\mathbb{E}[X]$ for every positive X .

Conditional expectation

In supervised learning, we want to predict a label Y from a description X . The natural object is then the average value of Y *when X is known*: this is the conditional expectation.

Definition 1.6: Conditional expectation

Let X be a discrete random variable and Y a random variable with $\mathbb{E}|Y| < +\infty$. For every x such that $\mathbb{P}(X = x) > 0$, the **conditional expectation of Y given $X = x$** is

$$\mathbb{E}[Y | X = x] = \frac{\mathbb{E}[Y \mathbb{1}_{\{X=x\}}]}{\mathbb{P}(X = x)} \left(= \sum_y y \mathbb{P}(Y = y | X = x) \text{ if } Y \text{ is discrete} \right).$$

Setting $g(x) = \mathbb{E}[Y | X = x]$, the **conditional expectation of Y given X** is the random variable $\mathbb{E}[Y | X] = g(X)$. When (X, Y) has a joint density $f_{X,Y}$, we define in the same way $g(x) = \int y f_{Y|X}(y | x) dy$ with the conditional density $f_{Y|X}(y | x) = f_{X,Y}(x, y) / f_X(x)$.

In words, $\mathbb{E}[Y | X = x]$ is the mean of Y over the sub-population of the outcomes for which $X = x$, and $\mathbb{E}[Y | X]$ is a *function of X* : it is the best guess of Y that we can make once we know X (we make this precise below). Conditional expectations are manipulated with the following rules.

Proposition 1.4: Properties of the conditional expectation

Let X, Y, Z be random variables with $\mathbb{E}|Y|, \mathbb{E}|Z| < +\infty$.

- (i) **Linearity**: $\mathbb{E}[aY + bZ | X] = a \mathbb{E}[Y | X] + b \mathbb{E}[Z | X]$.
- (ii) **Tower property (law of total expectation)**: $\mathbb{E}[\mathbb{E}[Y | X]] = \mathbb{E}[Y]$, i.e. $\mathbb{E}[Y] = \sum_x \mathbb{E}[Y | X = x] \mathbb{P}(X = x)$ in the discrete case.
- (iii) **Taking out what is known**: $\mathbb{E}[k(X)Y | X] = k(X) \mathbb{E}[Y | X]$ for every function k such that $\mathbb{E}|k(X)Y| < +\infty$.
- (iv) **Independence**: if X and Y are independent, then $\mathbb{E}[Y | X] = \mathbb{E}[Y]$.

Proof. In the discrete case. (i) is the linearity of the expectation applied to $Y \mathbb{1}_{\{X=x\}}$ and $Z \mathbb{1}_{\{X=x\}}$. (ii) Since the events $\{X = x\}$ form a partition of Ω , $\sum_x \mathbb{1}_{\{X=x\}} = 1$ and

$$\begin{aligned} \mathbb{E}[\mathbb{E}[Y | X]] &= \sum_x \mathbb{E}[Y | X = x] \mathbb{P}(X = x) = \sum_x \mathbb{E}[Y \mathbb{1}_{\{X=x\}}] \\ &= \mathbb{E}\left[Y \sum_x \mathbb{1}_{\{X=x\}}\right] = \mathbb{E}[Y]. \end{aligned}$$

(iii) On $\{X = x\}$, $k(X)Y = k(x)Y$, so $\mathbb{E}[k(X)Y \mid X = x] = k(x) \mathbb{E}[Y \mid X = x]$. (iv) By independence, $\mathbb{E}[Y \mathbb{1}_{\{X=x\}}] = \mathbb{E}[Y] \mathbb{P}(X = x)$. $\square$

The tower property is a “divide and conquer” rule: to compute an expectation, first average within each group defined by X , then average the group means, weighted by the size of the groups. Applied to $Y = \mathbb{1}_A$, it gives back the law of total probability (1.1).

Risks are expectations

We can now write the central objects of supervised learning in the language of probability; they are studied in detail in Section 6, where they are denoted by $\mathcal{R}^\ell(h)$ and $\mathcal{R}_S^\ell(h)$ (we drop the loss ℓ from the notation in this section). Let $\ell(h(\mathbf{x}), y)$ be the loss incurred by a predictor h on an example $(\mathbf{x}, y)$.

- The **true risk** of h is the expected loss on a new example drawn from $\mathcal{D}$:

$$\mathcal{R}(h) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(h(\mathbf{x}), y)].$$

For the 0-1 loss $\ell(h(\mathbf{x}), y) = \mathbb{1}_{\{h(\mathbf{x}) \neq y\}}$, Proposition 1.1 (iii) shows that $\mathcal{R}(h) = \mathbb{P}_{(\mathbf{x}, y) \sim \mathcal{D}} (h(\mathbf{x}) \neq y)$ is the probability of error of h .

- The **empirical risk** of h on the sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ is the average loss on the sample:

$$\mathcal{R}_S(h) = \frac{1}{m} \sum_{i=1}^m \ell(h(\mathbf{x}_i), y_i).$$

If h is *fixed*, i.e. chosen independently of S , the variables $Z_i = \ell(h(\mathbf{x}_i), y_i)$ are i.i.d. with mean $\mathcal{R}(h)$, and Theorems 1.1 and 1.2 give

$$\mathbb{E}_{S \sim \mathcal{D}^m} [\mathcal{R}_S(h)] = \mathcal{R}(h) \quad \text{and} \quad \text{Var}(\mathcal{R}_S(h)) = \frac{\text{Var}(Z_1)}{m} \leq \frac{(b-a)^2}{4m} \quad \text{if } \ell \in [a, b]. \quad (1.2)$$

The empirical risk is an *unbiased* estimator of the true risk, and its fluctuations decrease as $1/\sqrt{m}$. For the 0-1 loss, $m \mathcal{R}_S(h) \sim \text{Bin}(m, \mathcal{R}(h))$ is the number of errors on the sample.

Pitfall Equation (1.2) holds for a predictor fixed *before* seeing S . The predictor h_S returned by a learning algorithm depends on S , the losses $\ell(h_S(\mathbf{x}_i), y_i)$ are no longer independent of h_S , and in general $\mathbb{E}[\mathcal{R}_S(h_S)] < \mathbb{E}[\mathcal{R}(h_S)]$: the training error is optimistic. This is why we evaluate a model on a test set that was not used to learn it, and why generalization bounds must hold *uniformly* over all the predictors that the algorithm could have chosen (Section 1.4).

Conditional expectations also describe the best possible predictors, called *Bayes predictors*, which minimize the true risk over *all* functions; their risk is the Bayes risk $\mathcal{R}^*$ of Section 6.

Proposition 1.5: Bayes predictors

Let $(X, Y) \sim \mathcal{D}$.

- (i) **Regression with the squared loss.** Assume $\mathbb{E}[Y^2] < +\infty$ and let $f^*(x) = \mathbb{E}[Y | X = x]$ be the *regression function*. For every function g with $\mathbb{E}[g(X)^2] < +\infty$,

$$\mathbb{E}[(Y - g(X))^2] = \mathbb{E}[(Y - f^*(X))^2] + \mathbb{E}[(f^*(X) - g(X))^2].$$

Hence f^* minimizes the quadratic risk, and the excess risk of g is its mean squared distance to f^* .

- (ii) **Classification with the 0-1 loss.** Let $\eta_k(x) = \mathbb{P}(Y = k | X = x)$ for each class k . The classifier $h^*(x) \in \arg \max_k \eta_k(x)$ minimizes $\mathbb{P}(h(X) \neq Y)$ over all classifiers h , and its risk is $\mathcal{R}^* = 1 - \mathbb{E}[\max_k \eta_k(X)]$. For two classes with $\eta(x) = \mathbb{P}(Y = +1 | X = x)$, $h^*(x) = +1$ if and only if $\eta(x) \geq \frac{1}{2}$ (ties broken arbitrarily), and $\mathcal{R}^* = \mathbb{E}[\min(\eta(X), 1 - \eta(X))]$.

Proof. (i) Write $Y - g(X) = (Y - f^*(X)) + (f^*(X) - g(X))$ and expand the square. The cross term vanishes by the tower property and by taking out what is known:

$$\mathbb{E}[(Y - f^*(X))(f^*(X) - g(X))] = \mathbb{E}[(f^*(X) - g(X))(\mathbb{E}[Y | X] - f^*(X))] = 0.$$

- (ii) Conditionally on $X = x$, the classifier h errs unless $Y = h(x)$, so $\mathbb{P}(h(X) \neq Y | X = x) = 1 - \eta_{h(x)}(x) \geq 1 - \max_k \eta_k(x)$, with equality for $h = h^*$. Taking the expectation over X (tower property) gives $\mathbb{P}(h(X) \neq Y) \geq 1 - \mathbb{E}[\max_k \eta_k(X)] = \mathbb{P}(h^*(X) \neq Y)$. $\square$

In words, the best possible regressor predicts the mean of Y given X , and the best possible classifier predicts the most probable class given X . Neither can be computed in practice, since $\mathcal{D}$ is unknown: learning algorithms try to approximate them from data. The Bayes risk is not zero as soon as the label is not a deterministic function of $\mathbf{x}$ (two patients with the same description may have different diagnoses): it is the irreducible error of the problem.

1.3 Limit theorems: law of large numbers and central limit theorem

We now let the size of the sample grow. Let $X_1, X_2, \dots$ be i.i.d. random variables with mean μ and variance σ^2 , and let $S_m = \sum_{i=1}^m X_i$ and $\bar{X}_m = S_m/m$. We know that $\mathbb{E}[\bar{X}_m] = \mu$ and that $\text{Var}(\bar{X}_m) = \sigma^2/m \rightarrow 0$: the empirical mean becomes less and less random. The two classical limit theorems make this precise.

Theorem 1.6: Strong Law of Large Numbers; Kolmogorov, 1933

Let $(X_n)_{n \in \mathbb{N}^*}$ be a sequence of independent and identically distributed random variables, and let $S_n = \sum_{i=1}^n X_i$. Assume that they have a finite moment of order 1, i.e. $\mathbb{E}[|X_1|] < +\infty$.

Then the empirical mean S_n/n converges almost surely to $\mathbb{E}[X_1]$:

$$\mathbb{P}\left(\lim_{n \rightarrow +\infty} \frac{S_n}{n} = \mathbb{E}[X_1]\right) = 1.$$

The law of large numbers justifies the use of empirical averages (such as the empirical risk) to estimate expectations (such as the true risk), as well as the frequentist interpretation of probabilities: the frequency of an event in independent repetitions converges to its probability. It is illustrated in Fig. 3. A weaker version, the *weak law of large numbers*, states that $\mathbb{P}(|\bar{X}_m - \mu| \geq \varepsilon) \rightarrow 0$ for every $\varepsilon > 0$ (*convergence in probability*); we will prove it in two lines with Chebyshev's inequality (Corollary 1.11).

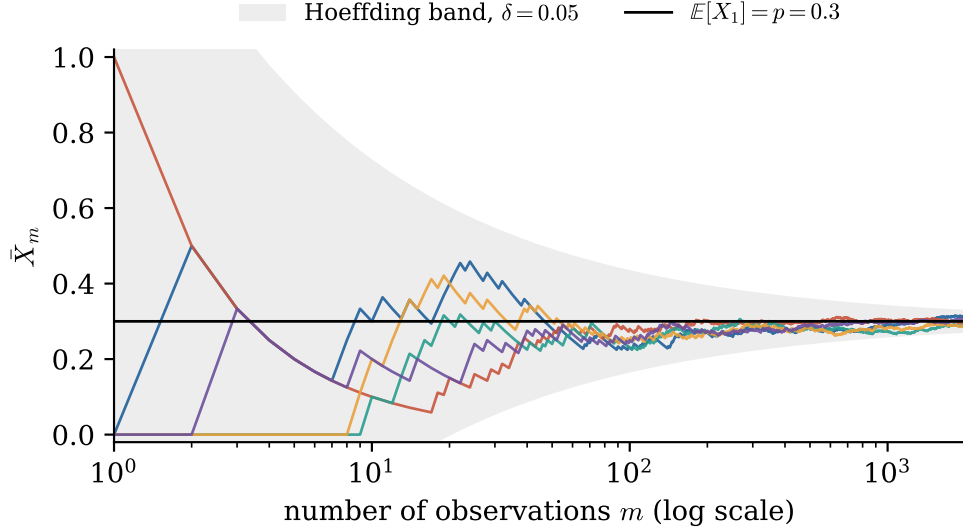


Figure 3: The law of large numbers: five trajectories $m \mapsto \bar{X}_m$ of the empirical mean of i.i.d. Bernoulli variables $\mathcal{B}(0.3)$ (think of the test error of a classifier with error rate 0.3, computed on more and more examples). All trajectories converge to 0.3, but for small m they wander far away from it. The grey band, given by Hoeffding's inequality (Corollary 1.15), contains $\bar{X}_m$ with probability at least 95% for each m . [code: make_figures_prob.py: fig_prob_lln]

The law of large numbers tells us that $\bar{X}_m - \mu \rightarrow 0$, but not at which speed. Since the standard deviation of $\bar{X}_m$ is $\sigma/\sqrt{m}$, it is natural to zoom in by a factor $\sqrt{m}$: the central limit theorem describes the distribution of the rescaled fluctuations.

Theorem 1.7: Central Limit Theorem

Let $(X_n)_{n \in \mathbb{N}^*}$ be i.i.d. random variables with mean μ and finite variance $\sigma^2 > 0$. Then $\sqrt{m}(\bar{X}_m - \mu)/\sigma$ converges in distribution to $\mathcal{N}(0, 1)$: for every $t \in \mathbb{R}$,

$$\lim_{m \rightarrow +\infty} \mathbb{P} \left(\frac{\sqrt{m}(\bar{X}_m - \mu)}{\sigma} \leq t \right) = \Phi(t) = \int_{-\infty}^t \frac{e^{-u^2/2}}{\sqrt{2\pi}} du.$$

Whatever the distribution of the X_i 's (provided that it has a finite variance), the average of many of them is approximately Gaussian, with mean μ and standard deviation $\sigma/\sqrt{m}$ (Fig. 4). This explains the ubiquity of the Gaussian distribution, and it gives the classical *asymptotic confidence interval*: since $\mathbb{P}(|Z| \leq 1.96) \approx 0.95$ for $Z \sim \mathcal{N}(0, 1)$, for large m ,

$$\mathbb{P} \left(|\bar{X}_m - \mu| \leq 1.96 \frac{\sigma}{\sqrt{m}} \right) \approx 0.95. \quad (1.3)$$

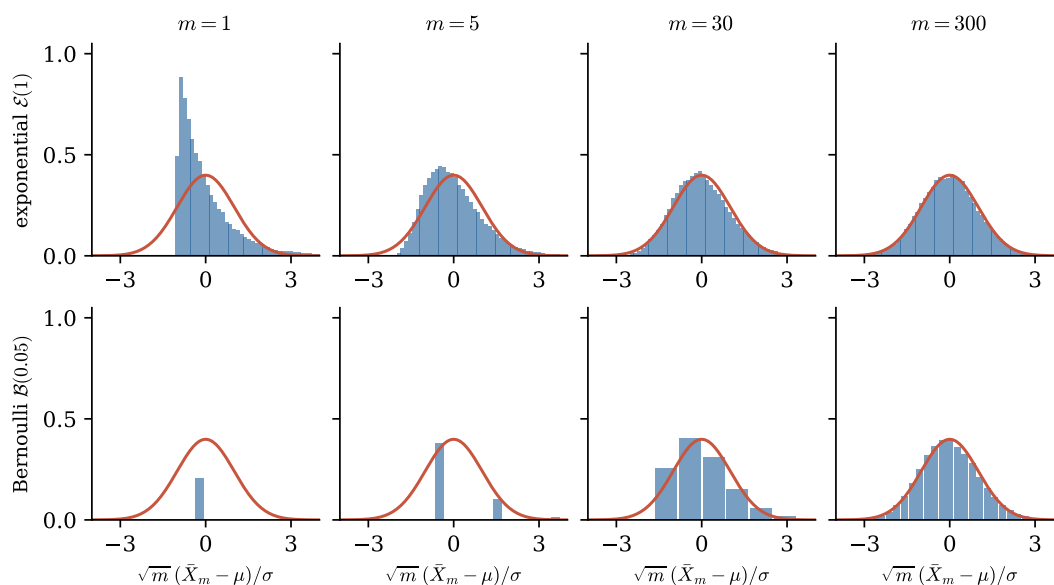


Figure 4: The central limit theorem: histograms of the standardized mean $\sqrt{m}(\bar{X}_m - \mu)/\sigma$ over 50 000 samples, for exponential variables (top) and for Bernoulli variables $\mathcal{B}(0.05)$ (bottom), compared with the density of $\mathcal{N}(0, 1)$ (red). For a skewed distribution the convergence is slow: for $m = 1$ and $m = 5$, the Bernoulli mean only takes a few values (for $m = 1$, the bar of the value $\bar{X}_1 = 1$ lies outside the window), and even for $m = 30$ the histogram is clearly asymmetric.

[code: make_figures_prob.py: fig_prob_clt]

Why limit theorems are not enough for learning

The limit theorems are statements about $m \rightarrow +\infty$; a learning algorithm works with a fixed, finite sample. This makes them insufficient for our purpose, for several reasons.

- **They are asymptotic.** The law of large numbers says nothing about $\bar{X}_{1000}$, and the central limit theorem says that the Gaussian approximation is good *for m large enough*, without telling us how large. The answer depends on the distribution: in Fig. 4, $m = 30$ is enough for exponential variables but not for Bernoulli variables with a small parameter. Yet, a classifier with a small error rate is precisely what we hope to learn: for $p = 0.01$ and $m = 100$, the “95%” interval (1.3) computed with the estimated variance contains p only 63% of the time.
- **They are inaccurate for small probabilities.** Even when the Gaussian approximation is good in the bulk of the distribution, its *relative* error in the tails can be large. Yet, we need very small probabilities as soon as we want guarantees for many predictors at once: with the union bound, certifying 1000 classifiers at confidence 95% requires each individual statement to fail with probability at most 5×10^{-5} .
- **They involve unknown quantities**, such as σ , which must itself be estimated.

What we need are *non-asymptotic* statements: explicit upper bounds on $\mathbb{P}(|\bar{X}_m - \mu| \geq \varepsilon)$, valid for every m and every distribution in a given class (e.g. bounded variables). These are the concentration inequalities.

Historical note The first law of large numbers was proved by Jacob Bernoulli for coin tosses, in his *Ars Conjectandi* published in 1713; de Moivre (1733) and Laplace (1810s) then showed that the binomial distribution is approximately Gaussian, the first form of the central limit theorem. Probability theory became a branch of mathematics in its own right with the axioms of Kolmogorov in 1933 [Kolmogorov \(1933\)](#), which are the ones of Definition 1.1; Kolmogorov also proved the strong law of large numbers in its general form.

1.4 Concentration inequalities

A concentration inequality bounds the probability that a random variable (typically an average) deviates from its expectation by more than a given amount. We present them from the crudest to the sharpest; remarkably, all of them derive from the first one, Markov's inequality, applied to well-chosen functions of the variable. We start with an even simpler tool, which is used to combine several bounds.

The union bound

Proposition 1.8: Union bound (Boole's inequality)

For every finite or countable family of events $(A_k)_k$,

$$\mathbb{P}\left(\bigcup_k A_k\right) \leq \sum_k \mathbb{P}(A_k).$$

Proof. For every outcome, the number of events A_k that occur is at least 1 if at least one of them occurs, so $\mathbb{1}_{\bigcup_k A_k} \leq \sum_k \mathbb{1}_{A_k}$. Taking expectations (monotonicity, linearity, and monotone convergence for a countable sum) gives the result. $\square$

The union bound is crude, since it counts several times the outcomes where several events occur, but it requires no assumption at all on the dependence between the events. It is used as follows: if each of K statements fails with probability at most δ/K , then, with probability at least $1 - \delta$, *all* of them hold simultaneously. This is the simplest way to obtain guarantees that are valid for all the predictors of a finite class at once ([Section 1.4](#)).

Markov's and Chebyshev's inequalities

Proposition 1.9: Markov's Inequality

Let X be a nonnegative random variable. Then, for all $t > 0$,

$$\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}[X]}{t}.$$

Equivalently, for every $\delta \in (0, 1]$, with probability at least $1 - \delta$, $X < \mathbb{E}[X]/\delta$.

Proof. Since $X \geq 0$, we have $X \geq t \mathbb{1}_{\{X \geq t\}}$: when $X \geq t$, the right-hand side equals $t \leq X$, and otherwise it is $0 \leq X$. Taking the expectation (Proposition 1.1 (ii) and (iii)) gives $\mathbb{E}[X] \geq t \mathbb{P}(X \geq t)$. The second form follows with $t = \mathbb{E}[X]/\delta$. $\square$

Markov's inequality only uses the mean, and it cannot be improved without further information: for the variable X equal to t with probability $\mathbb{E}[X]/t$ and to 0 otherwise, it is an equality. In words, a nonnegative variable cannot often be much larger than its mean: at most 10% of a population can earn more than ten times the average income. It is weak (it decreases only as $1/t$), but it becomes powerful when applied to $g(X)$ for a well-chosen nonnegative increasing function g , since $\mathbb{P}(X \geq t) \leq \mathbb{P}(g(X) \geq g(t)) \leq \mathbb{E}[g(X)]/g(t)$. With $g(x) = x^2$, we get Chebyshev's inequality; with $g(x) = e^{\lambda x}$, Chernoff's method.

Proposition 1.10: Chebyshev's (Bienaymé–Chebyshev) Inequality

Let X be a random variable with a finite variance $\text{Var}(X)$. Then, for all $\varepsilon > 0$,

$$\mathbb{P}(|X - \mathbb{E}[X]| \geq \varepsilon) \leq \frac{\text{Var}(X)}{\varepsilon^2}.$$

Proof. Apply Markov's inequality to the nonnegative random variable $(X - \mathbb{E}[X])^2$ with $t = \varepsilon^2$:

$$\mathbb{P}(|X - \mathbb{E}[X]| \geq \varepsilon) = \mathbb{P}((X - \mathbb{E}[X])^2 \geq \varepsilon^2) \leq \frac{\mathbb{E}[(X - \mathbb{E}[X])^2]}{\varepsilon^2} = \frac{\text{Var}(X)}{\varepsilon^2}.$$

$\square$

Corollary 1.11: Weak law of large numbers

Let $X_1, \dots, X_m$ be pairwise uncorrelated (e.g. independent) random variables with the same mean μ and the same variance σ^2 . Then, for every $\varepsilon > 0$,

$$\mathbb{P}(|\bar{X}_m - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{m\varepsilon^2} \xrightarrow{m \rightarrow +\infty} 0.$$

Proof. Apply Chebyshev's inequality to $\bar{X}_m$, whose variance is σ^2/m (Proposition 1.2). $\square$

Example 1.4 (Running question, first answer). A classifier is evaluated on m test examples; the losses are i.i.d. $\mathcal{B}(p)$, with variance $p(1-p) \leq \frac{1}{4}$. Setting $\frac{1}{4m\varepsilon^2} = \delta$, Chebyshev's inequality shows that, with probability at least $1 - \delta$, $|\hat{p} - p| < \frac{1}{2\sqrt{m\delta}}$. To get a precision $\varepsilon = 0.05$ with confidence 95% ($\delta = 0.05$), it suffices to have $m \geq \frac{1}{4\delta\varepsilon^2} = 2000$ test examples; for a confidence of 99.9%, we need $m \geq 100\,000$.

The dependence in $1/\delta$ of Chebyshev's sample size is a problem: high confidence becomes very expensive. The reason is that Chebyshev's inequality only uses the variance, and must therefore hold for distributions with heavy tails. For bounded variables, much more is true.

Chernoff's method

The idea of Chernoff's method is to apply Markov's inequality to the exponential $e^{\lambda X}$, and then to choose the best value of the free parameter λ . The function $\lambda \mapsto \mathbb{E}[e^{\lambda X}]$ is the *moment generating function* (MGF) of X .

Proposition 1.12: Chernoff's bound

For every random variable X and every $t \in \mathbb{R}$,

$$\mathbb{P}(X \geq t) \leq \inf_{\lambda \geq 0} e^{-\lambda t} \mathbb{E}[e^{\lambda X}].$$

Proof. For $\lambda = 0$, the right-hand side equals 1. For $\lambda > 0$, the function $x \mapsto e^{\lambda x}$ is increasing and positive, so Markov's inequality gives $\mathbb{P}(X \geq t) = \mathbb{P}(e^{\lambda X} \geq e^{\lambda t}) \leq e^{-\lambda t} \mathbb{E}[e^{\lambda X}]$. The bound holds for every $\lambda \geq 0$, hence for the infimum. $\square$

Chernoff's method is particularly well suited to sums of independent variables, because the MGF of a sum of independent variables is the product of their MGFs (Proposition 1.1 (iv)): bounding the MGF of each term is enough.

Example 1.5 (Gaussian tails). If $X \sim \mathcal{N}(0, \sigma^2)$, then $\mathbb{E}[e^{\lambda X}] = e^{\lambda^2 \sigma^2 / 2}$, so $\mathbb{P}(X \geq t) \leq \inf_{\lambda \geq 0} e^{-\lambda t + \lambda^2 \sigma^2 / 2} = e^{-t^2 / (2\sigma^2)}$ for $t \geq 0$ (the infimum is reached at $\lambda = t / \sigma^2$). The tail decreases like $e^{-t^2 / (2\sigma^2)}$, much faster than the σ^2 / t^2 of Chebyshev.

Example 1.6 (Bernoulli variables and the binary kl). Let $X_1, \dots, X_m$ be i.i.d. $\mathcal{B}(p)$ and $q \in (p, 1)$. Since $\mathbb{E}[e^{\lambda X_i}] = 1 - p + pe^\lambda$, Chernoff's bound applied to $S_m = \sum_i X_i$ and $t = mq$ gives

$$\mathbb{P}(\bar{X}_m \geq q) \leq \inf_{\lambda \geq 0} e^{-\lambda mq} (1 - p + pe^\lambda)^m = \exp\left(-m \sup_{\lambda \geq 0} [\lambda q - \ln(1 - p + pe^\lambda)]\right).$$

The supremum is reached where the derivative vanishes, i.e. at $e^\lambda = \frac{q(1-p)}{p(1-q)}$, and a direct computation shows that its value is the **binary Kullback–Leibler divergence**

$$\text{kl}(q||p) = q \ln \frac{q}{p} + (1 - q) \ln \frac{1 - q}{1 - p}.$$

Hence $\mathbb{P}(\bar{X}_m \geq q) \leq e^{-m \text{kl}(q||p)}$ for $q > p$, and symmetrically $\mathbb{P}(\bar{X}_m \leq q) \leq e^{-m \text{kl}(q||p)}$ for $q < p$. This is the sharpest bound that can be obtained by this method, and it is the form used in PAC-Bayesian theory (the kl test-set bound, Theorem 13.5 in Section 13).

Hoeffding's inequality

For a general bounded variable, we do not know its MGF, but we can bound it.

Lemma 1.13: Hoeffding's Lemma

Let X be a random variable such that $a \leq X \leq b$ almost surely. Then, for all $s \in \mathbb{R}$,

$$\mathbb{E} \left[e^{s(X - \mathbb{E}[X])} \right] \leq \exp \left(\frac{s^2(b-a)^2}{8} \right).$$

Proof. Replacing X by $X - \mathbb{E}[X]$ (and a, b by $a - \mathbb{E}[X], b - \mathbb{E}[X]$), we can assume that $\mathbb{E}[X] = 0$, so that $a \leq 0 \leq b$; we can also assume $a < b$, the case $a = b$ being trivial. Let $s \in \mathbb{R}$.

Step 1: convexity. Every $x \in [a, b]$ is the convex combination $x = \frac{b-x}{b-a}a + \frac{x-a}{b-a}b$, so the convexity of $x \mapsto e^{sx}$ gives $e^{sx} \leq \frac{b-x}{b-a}e^{sa} + \frac{x-a}{b-a}e^{sb}$. Applying it to X and taking the expectation, with $\mathbb{E}[X] = 0$,

$$\mathbb{E} [e^{sX}] \leq \frac{b}{b-a}e^{sa} - \frac{a}{b-a}e^{sb} = (1-\theta)e^{sa} + \theta e^{sb}, \quad \text{where } \theta = \frac{-a}{b-a} \in [0, 1].$$

Step 2: change of variable. Let $u = s(b-a)$. Since $sa = -\theta u$, the right-hand side equals $e^{-\theta u}(1-\theta + \theta e^u) = e^{\varphi(u)}$, with

$$\varphi(u) = -\theta u + \ln(1-\theta + \theta e^u).$$

Step 3: Taylor expansion. We have $\varphi(0) = 0$, $\varphi'(u) = -\theta + \frac{\theta e^u}{1-\theta + \theta e^u}$, so $\varphi'(0) = 0$, and

$$\varphi''(u) = \frac{\theta e^u(1-\theta)}{(1-\theta + \theta e^u)^2} = \rho(1-\rho) \leq \frac{1}{4}, \quad \text{where } \rho = \frac{\theta e^u}{1-\theta + \theta e^u} \in [0, 1].$$

By Taylor's theorem, there exists ξ between 0 and u such that $\varphi(u) = \varphi(0) + u\varphi'(0) + \frac{u^2}{2}\varphi''(\xi) \leq \frac{u^2}{8} = \frac{s^2(b-a)^2}{8}$. $\square$

The lemma says that a bounded centred variable has an MGF dominated by that of a Gaussian variable with variance $(b-a)^2/4$: bounded variables have (at least) Gaussian tails. Combined with Chernoff's method, it gives the most used concentration inequality of learning theory.

Proposition 1.14: Hoeffding's Inequality

Let $X_1, \dots, X_n$ be independent random variables such that, for all i , $a_i \leq X_i \leq b_i$ almost surely, where $a_i < b_i$ are real numbers, and let $S_n = \sum_{i=1}^n X_i$. Then, for all $\varepsilon > 0$,

$$\mathbb{P}(S_n - \mathbb{E}[S_n] \geq \varepsilon) \leq \exp \left(-\frac{2\varepsilon^2}{\sum_{i=1}^n (b_i - a_i)^2} \right),$$

and the same bound holds for $\mathbb{P}(S_n - \mathbb{E}[S_n] \leq -\varepsilon)$. Consequently,

$$\mathbb{P}(|S_n - \mathbb{E}[S_n]| \geq \varepsilon) \leq 2 \exp \left(-\frac{2\varepsilon^2}{\sum_{i=1}^n (b_i - a_i)^2} \right).$$

In particular, if all the X_i 's take their values in $[a, b]$, then for the empirical mean

$$\bar{X}_n = S_n/n,$$

$$\mathbb{P}\left(|\bar{X}_n - \mathbb{E}[\bar{X}_n]| \geq \varepsilon\right) \leq 2 \exp\left(-\frac{2n\varepsilon^2}{(b-a)^2}\right).$$

Proof. Let $s > 0$ and $c^2 = \sum_{i=1}^n (b_i - a_i)^2$. By Chernoff's bound (Proposition 1.12) applied to $S_n - \mathbb{E}[S_n]$,

$$\begin{aligned} \mathbb{P}(S_n - \mathbb{E}[S_n] \geq \varepsilon) &\leq e^{-s\varepsilon} \mathbb{E}\left[e^{s(S_n - \mathbb{E}[S_n])}\right] = e^{-s\varepsilon} \mathbb{E}\left[\prod_{i=1}^n e^{s(X_i - \mathbb{E}[X_i])}\right] \\ &\downarrow \text{independence, Proposition 1.1 (iv)} \\ &= e^{-s\varepsilon} \prod_{i=1}^n \mathbb{E}\left[e^{s(X_i - \mathbb{E}[X_i])}\right] \\ &\downarrow \text{Hoeffding's lemma} \\ &\leq \exp\left(-s\varepsilon + \frac{s^2 c^2}{8}\right). \end{aligned}$$

The exponent is a quadratic function of s , minimal for $s = 4\varepsilon/c^2$, where it equals $-2\varepsilon^2/c^2$: this gives the first bound. The lower tail is obtained by applying this result to the variables $-X_i \in [-b_i, -a_i]$, and the two-sided bound by a union bound on the two events. For the empirical mean, $\{|\bar{X}_n - \mathbb{E}[\bar{X}_n]| \geq \varepsilon\} = \{|S_n - \mathbb{E}[S_n]| \geq n\varepsilon\}$ and $c^2 = n(b-a)^2$. $\square$

Figure 5 compares the inequalities seen so far with the exact tail of a binomial distribution. The polynomial bounds of Markov and Chebyshev are useless for small probabilities, while the exponential bounds of Hoeffding and Chernoff decrease at the right speed, Chernoff's kl bound being the closest to the truth.

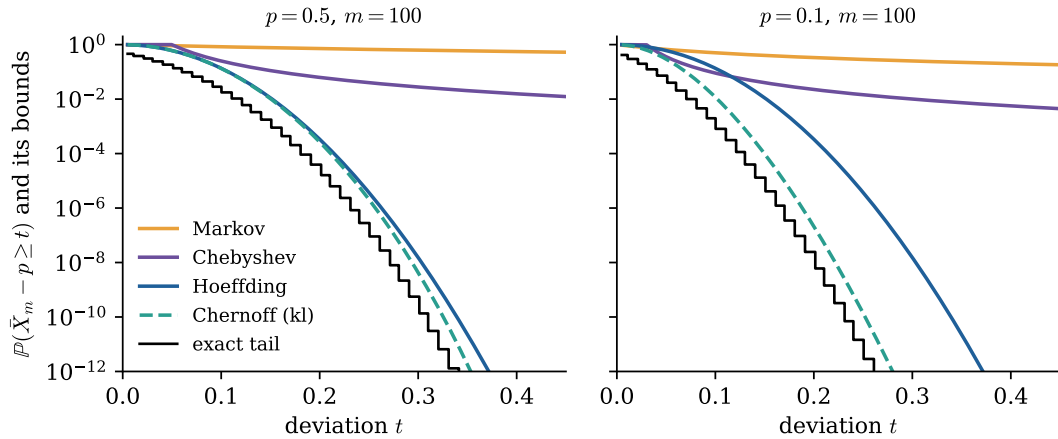


Figure 5: Upper tail $\mathbb{P}(\bar{X}_m - p \geq t)$ of the mean of $m = 100$ Bernoulli variables (black, exact binomial tail) and its upper bounds (log scale): Markov ($p/(p+t)$), Chebyshev ($p(1-p)/(mt^2)$), Hoeffding (e^{-2mt^2}) and Chernoff ($e^{-m \text{kl}(p+t||p)}$). For $p = 0.5$ (left), Hoeffding's and Chernoff's bounds almost coincide; for $p = 0.1$ (right), the variance $p(1-p)$ is small, and the bounds that adapt to it (Chernoff, and Chebyshev for small t) are much better than Hoeffding's. [code: make_figures_prob.py: fig_prob_tailbounds]

From a tail bound to a confidence statement

A concentration inequality is usually stated as “the probability of a deviation larger than ε is at most $g(\varepsilon)$ ”. In learning theory, we prefer the inverse form “with probability at least $1 - \delta$, the deviation is at most $\varepsilon(\delta)$ ”. To go from one to the other, we solve $g(\varepsilon) = \delta$ in ε .

Corollary 1.15: Hoeffding’s inequality, inverse form; sample size

Let $X_1, \dots, X_m$ be independent random variables with values in $[a, b]$ and the same mean μ , and let $\delta \in (0, 1)$. With probability at least $1 - \delta$,

$$|\bar{X}_m - \mu| < (b - a) \sqrt{\frac{\ln(2/\delta)}{2m}},$$

and (one-sided version), with probability at least $1 - \delta$, $\mu < \bar{X}_m + (b - a) \sqrt{\frac{\ln(1/\delta)}{2m}}$. Consequently, to estimate μ with precision ε and confidence $1 - \delta$, it suffices to have

$$m \geq \frac{(b - a)^2 \ln(2/\delta)}{2\varepsilon^2}.$$

Proof. Set $2e^{-2m\varepsilon^2/(b-a)^2} = \delta$, i.e. $\varepsilon = (b - a) \sqrt{\ln(2/\delta)/(2m)}$, in Proposition 1.14: the probability of the event $\{|\bar{X}_m - \mu| \geq \varepsilon\}$ is at most δ , so its complement has probability at least $1 - \delta$. The one-sided statement is obtained in the same way from the bound on the lower tail $\mathbb{P}(\bar{X}_m - \mu \leq -\varepsilon)$, without the factor 2. The sample size is the smallest m for which $(b - a) \sqrt{\ln(2/\delta)/(2m)} \leq \varepsilon$. $\square$

Example 1.7 (Running question, second answer). For the test error of a fixed classifier ($a = 0, b = 1$), Hoeffding’s inequality guarantees $|\hat{p} - p| < 0.05$ with confidence 95% as soon as $m \geq \frac{\ln 40}{2 \times 0.05^2} \approx 737.8$, i.e. $m = 738$, instead of 2000 with Chebyshev; with confidence 99.9%, we need $m = 1521$ instead of 100 000. With $m = 1000$ test examples and an observed error $\hat{p} = 0.12$, we can claim that $p \in [0.077, 0.163]$ with confidence 95%, since $\sqrt{\ln 40/2000} \approx 0.043$.

The comparison is summarized in Fig. 6. All the bounds decrease as $1/\sqrt{m}$, but the price of confidence is very different: Chebyshev’s sample size grows as $1/\delta$, Hoeffding’s only as $\ln(1/\delta)$. This logarithmic dependence is what makes the union bound affordable: replacing δ by δ/K only adds $\ln K$ to the numerator.

Pitfall In “with probability at least $1 - \delta$, $p \in [\hat{p} - \varepsilon, \hat{p} + \varepsilon]$ ”, the randomness is in the *sample*, hence in $\hat{p}$, not in p , which is a fixed unknown number. The statement means: if we repeated the whole experiment many times, the interval would contain p in at least $100(1 - \delta)\%$ of the repetitions. Once the sample has been observed, the interval $[0.077, 0.163]$ either contains p or not; we simply do not know which, and our procedure fails at most 5% of the time.

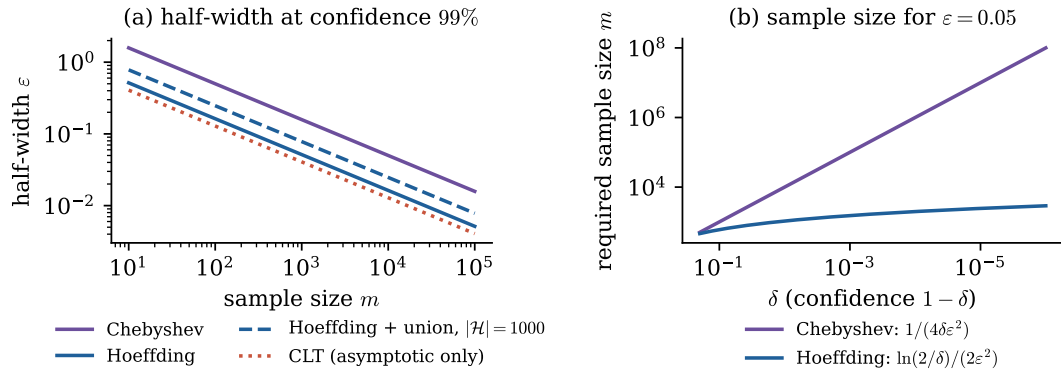


Figure 6: (a) Half-width ε of a confidence interval at level 99% for the mean of variables with values in $[0, 1]$, as a function of m (log-log scale): all decrease as $1/\sqrt{m}$, and the union bound over 1000 hypotheses (Section 1.4) costs less than Chebyshev’s inequality for a single one. The CLT curve is shown for comparison, with the worst-case variance $\frac{1}{4}$; it is only an approximation. (b) Sample size needed for a precision $\varepsilon = 0.05$, as a function of δ : $1/\delta$ for Chebyshev versus $\ln(2/\delta)$ for Hoeffding. [code: make_figures_prob.py: fig_prob_ciwidth]

Bernstein’s inequality: adapting to the variance

Hoeffding’s inequality only uses the range $[a, b]$ of the variables. It is therefore tight for the worst case, Bernoulli variables with $p = \frac{1}{2}$, whose variance is $\frac{1}{4}$, but pessimistic when the variance is much smaller than $(b - a)^2/4$, which is the case of the test error of a good classifier: a Bernoulli variable with mean 0.01 has variance 0.0099. Bernstein’s inequality takes the variance into account.

Proposition 1.16: Bernstein’s inequality

Let $X_1, \dots, X_m$ be independent random variables with the same mean μ , such that $|X_i - \mu| \leq c$ almost surely, and let $\sigma^2 = \frac{1}{m} \sum_{i=1}^m \text{Var}(X_i)$. Then, for every $\varepsilon > 0$,

$$\mathbb{P}(\bar{X}_m - \mu \geq \varepsilon) \leq \exp\left(-\frac{m\varepsilon^2}{2\sigma^2 + 2c\varepsilon/3}\right),$$

and the same bound holds for $\mathbb{P}(\bar{X}_m - \mu \leq -\varepsilon)$. Equivalently, for every $\delta \in (0, 1)$, with probability at least $1 - \delta$,

$$\bar{X}_m - \mu < \sqrt{\frac{2\sigma^2 \ln(1/\delta)}{m}} + \frac{2c \ln(1/\delta)}{3m}.$$

The proof follows Chernoff’s method, with a bound on the MGF that involves the variance instead of the range (see Boucheron et al., 2013, Chapter 2). The inverse form is obtained by solving a quadratic equation in ε and using $\sqrt{u+v} \leq \sqrt{u} + \sqrt{v}$. Let us read it.

- For small deviations ($\varepsilon \ll \sigma^2/c$), the exponent is about $m\varepsilon^2/(2\sigma^2)$: the bound behaves like the Gaussian approximation of the central limit theorem, with the *true* variance. Since $\sigma^2 \leq c^2$, and for a Bernoulli variable $\sigma^2 = p(1-p) \leq \frac{1}{4}$, this

is never much worse than Hoeffding, and much better when σ^2 is small.

- For a small variance, the second term dominates, and the deviation is of order $\ln(1/\delta)/m$ instead of $\sqrt{\ln(1/\delta)/m}$: this is a *fast rate*. For a classifier with error rate $p = 0.01$, $m = 1000$ and $\delta = 0.05$, Bernstein's one-sided deviation is 0.0097, four times smaller than Hoeffding's 0.0387.
- The price is that σ^2 is unknown in general. In practice, it is either bounded (for the 0-1 loss, $\sigma^2 = p(1-p) \leq p$, which leads to bounds that are implicit in p) or replaced by the empirical variance (*empirical Bernstein* inequalities [Maurer and Pontil \(2009\)](#)). The kl bound of [example 1.6](#) adapts to the variance automatically, and this is one reason why it is preferred in Part IV.

McDiarmid's inequality: beyond sums

Hoeffding's inequality concerns sums of independent variables. McDiarmid's inequality extends it to any function of independent variables that does not depend too much on each of them; it is used, for instance, to derive generalization bounds based on the Rademacher complexity or on algorithmic stability.

Proposition 1.17: McDiarmid's (Bounded Differences) Inequality

Let $X_1, \dots, X_n$ be independent random variables taking values in a set $\mathcal{X}$, and let $f : \mathcal{X} \rightarrow \mathbb{R}$ be a function satisfying the **bounded differences property**: there exist constants $c_1, \dots, c_n \geq 0$ such that, for all i and all $x_1, \dots, x_n, x'_i \in \mathcal{X}$,

$$|f(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_{i-1}, x'_i, x_{i+1}, \dots, x_n)| \leq c_i.$$

Then, for all $\varepsilon > 0$,

$$\mathbb{P}\left(f(X_1, \dots, X_n) - \mathbb{E}[f(X_1, \dots, X_n)] \geq \varepsilon\right) \leq \exp\left(-\frac{2\varepsilon^2}{\sum_{i=1}^n c_i^2}\right),$$

and the same bound holds for the lower tail, so that the two-sided deviation is bounded by twice this quantity.

Hoeffding's inequality is the special case $f(x_1, \dots, x_n) = \sum_{i=1}^n x_i$ with $x_i \in [a_i, b_i]$, for which $c_i = b_i - a_i$. The proof of McDiarmid's inequality applies the Chernoff method and Hoeffding's lemma to the martingale differences $V_i = \mathbb{E}[f \mid X_1, \dots, X_i] - \mathbb{E}[f \mid X_1, \dots, X_{i-1}]$, each of which lies, conditionally on $X_1, \dots, X_{i-1}$, in an interval of length at most c_i ; since $f - \mathbb{E}[f] = \sum_i V_i$, the argument is the same as for a sum (see [McDiarmid, 1989](#); [Boucheron et al., 2013](#)).

The strength of McDiarmid's inequality is that f can be very complicated, as long as no single example has too much influence on it. Here is the application that matters in learning theory. Let $\mathcal{H}$ be a class of predictors and ℓ a loss with values in $[0, 1]$, and consider the worst deviation between the true and the empirical risks over the class,

$$\Phi(S) = \sup_{h \in \mathcal{H}} (\mathcal{R}(h) - \mathcal{R}_S(h)).$$

If we replace the example $(\mathbf{x}_i, y_i)$ of S by another one, the empirical risk of each h changes by at most $1/m$ (a single term of the average changes, and it lies in $[0, 1]$), hence so does the supremum. McDiarmid's inequality with $c_i = 1/m$ then shows that, with probability at least $1 - \delta$,

$$\forall h \in \mathcal{H}, \quad \mathcal{R}(h) \leq \mathcal{R}_S(h) + \Phi(S) \leq \mathcal{R}_S(h) + \mathbb{E}_{S \sim \mathcal{D}^m}[\Phi(S)] + \sqrt{\frac{\ln(1/\delta)}{2m}}.$$

A random quantity, the worst deviation over $\mathcal{H}$, has been replaced by its expectation, up to a Hoeffding-like term. What remains is to bound this expectation: this is exactly what the Rademacher complexity does (Definition 10.1 and Theorem 10.2 in Section 6 and the following ones), and it is how the bounds based on algorithmic stability are proved.

Preview: uniform bounds and PAC-Bayes

Let us conclude by showing how the tools of this section combine into the generalization bounds of the rest of the course.

A finite class of predictors. Let $\mathcal{H} = \{h_1, \dots, h_K\}$ and a loss with values in $[0, 1]$. For each fixed h_k , Hoeffding's inequality gives $\mathbb{P}(|\mathcal{R}(h_k) - \mathcal{R}_S(h_k)| \geq \varepsilon) \leq 2e^{-2m\varepsilon^2}$. By the union bound,

$$\mathbb{P}_{S \sim \mathcal{D}^m}(\exists h \in \mathcal{H} : |\mathcal{R}(h) - \mathcal{R}_S(h)| \geq \varepsilon) \leq \sum_{k=1}^K \mathbb{P}(|\mathcal{R}(h_k) - \mathcal{R}_S(h_k)| \geq \varepsilon) \leq 2Ke^{-2m\varepsilon^2}.$$

Setting the right-hand side to δ : with probability at least $1 - \delta$,

$$\forall h \in \mathcal{H}, \quad |\mathcal{R}(h) - \mathcal{R}_S(h)| < \sqrt{\frac{\ln K + \ln(2/\delta)}{2m}}. \quad (1.4)$$

Since the bound holds simultaneously for all $h \in \mathcal{H}$, it holds in particular for the predictor h_S chosen by the learning algorithm, whatever the way it uses S . The price of uniformity is the term $\ln K = \ln |\mathcal{H}|$, which measures the complexity of the class; Fig. 7 shows that it is real, not an artefact of the proof. This is the starting point of Section 6, which then replaces $\ln |\mathcal{H}|$ by the VC-dimension or the Rademacher complexity for infinite classes.

The kl version. Replacing Hoeffding's inequality by Chernoff's kl bound of example 1.6 gives, for the 0-1 loss and with probability at least $1 - \delta$, $\text{kl}(\mathcal{R}_S(h) \parallel \mathcal{R}(h)) \leq \frac{\ln K + \ln(1/\delta)}{m}$ for every $h \in \mathcal{H}$ such that $\mathcal{R}_S(h) \leq \mathcal{R}(h)$. This bound adapts to the variance, like Bernstein's inequality: when the empirical risk is close to 0, it gives a deviation of order $\frac{\ln K + \ln(1/\delta)}{m}$ instead of its square root. It is the *Occam bound* of Section 13.

Change of measure. PAC-Bayesian theory (Part IV) goes one step further: instead of a single predictor, it certifies a *distribution* Q over $\mathcal{H}$ (for instance, the weights of the voters of a majority vote). Its key tool is a direct consequence of Jensen's inequality. For

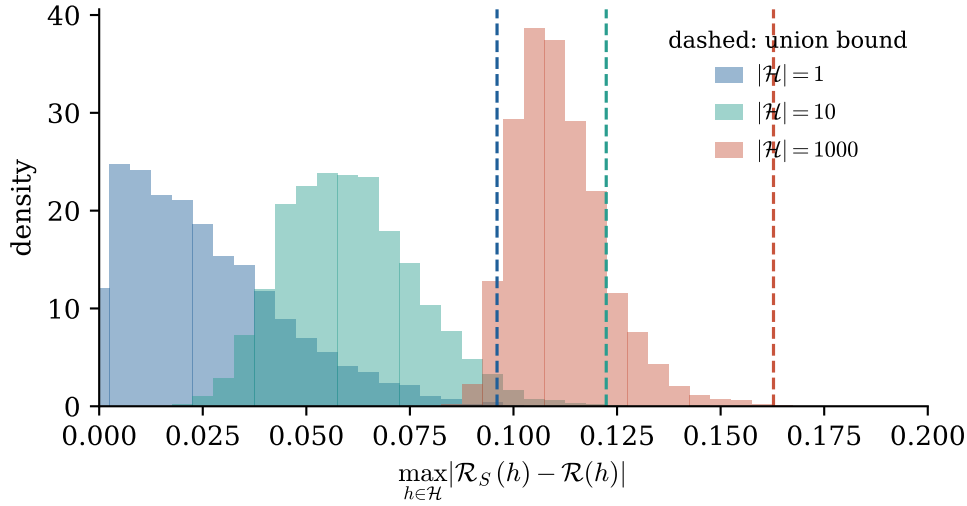


Figure 7: The maximal deviation $\max_{h \in \mathcal{H}} |\mathcal{R}_S(h) - \mathcal{R}(h)|$ over $K = |\mathcal{H}|$ classifiers with true risk 0.3 whose errors are independent, for $m = 200$ (4000 simulated samples). The more classifiers, the larger the worst deviation: picking the classifier with the smallest empirical risk among many is optimistic. The dashed lines show the uniform bound (1.4) with $\delta = 0.05$, which grows only like $\sqrt{\ln K}$. [code: make_figures_prob.py: fig_prob_union]

two distributions P, Q on a finite set $\mathcal{H}$, with $P(h) > 0$ for all h , the *Kullback–Leibler divergence* is $\text{KL}(Q\|P) = \sum_h Q(h) \ln \frac{Q(h)}{P(h)} \geq 0$, and for every function $\phi : \mathcal{H} \rightarrow \mathbb{R}$,

$$\mathbb{E}_{h \sim Q}[\phi(h)] \leq \text{KL}(Q\|P) + \ln \mathbb{E}_{h \sim P}[e^{\phi(h)}]. \quad (1.5)$$

Indeed, summing over the h such that $Q(h) > 0$ and using Jensen’s inequality for the concave function $\ln$,

$$\begin{aligned} \mathbb{E}_{h \sim Q}[\phi(h)] - \text{KL}(Q\|P) &= \mathbb{E}_{h \sim Q} \left[\ln \left(e^{\phi(h)} \frac{P(h)}{Q(h)} \right) \right] \leq \ln \mathbb{E}_{h \sim Q} \left[e^{\phi(h)} \frac{P(h)}{Q(h)} \right] \\ &= \ln \sum_{h: Q(h) > 0} P(h) e^{\phi(h)} \leq \ln \mathbb{E}_{h \sim P}[e^{\phi(h)}]. \end{aligned}$$

A PAC-Bayesian bound is then obtained in three steps that use the whole toolbox of this section: (1) bound the exponential moment $\mathbb{E}_S[e^{\phi_S(h)}]$ of a deviation $\phi_S(h)$ of a fixed h (Hoeffding’s lemma, or Chernoff’s kl bound); (2) average it over the prior P , exchange the expectations, and apply Markov’s inequality to the nonnegative variable $\mathbb{E}_{h \sim P}[e^{\phi_S(h)}]$; (3) transfer the result to any posterior Q with (1.5). This is done step by step in Section 14 (Theorem 14.4).

Historical note Chebyshev’s inequality was published by Bienaymé in 1853 and, independently, by Chebyshev in 1867; Markov, a student of Chebyshev, used the simpler inequality that now bears his name a few years later. Bernstein’s inequality dates back to the 1920s. The exponential method was popularized in statistics by Chernoff [Chernoff \(1952\)](#), and Hoeffding’s inequality was published in 1963 [Hoeffding \(1963\)](#). McDiarmid’s survey [McDiarmid \(1989\)](#) made the bounded differences inequality popular in combinatorics and computer science, and it soon became a standard tool of learning theory. The book of Boucheron, Lugosi and

Massart [Boucheron et al. \(2013\)](#) is the reference on modern concentration inequalities.

To remember from Section 1

- The data are modelled as an i.i.d. sample $S \sim \mathcal{D}^m$; every quantity computed from S is a random variable.
- The true risk $\mathcal{R}(h) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}}[\ell(h(\mathbf{x}), y)]$ is an expectation; for a fixed h , the empirical risk is an unbiased average whose standard deviation decreases as $1/\sqrt{m}$. The Bayes predictors are conditional expectations (regression) and most probable classes (classification).
- The expectation is linear; Jensen's inequality orders $f(\mathbb{E}[X])$ and $\mathbb{E}[f(X)]$ for a convex or concave f ; the tower property computes expectations by conditioning.
- The law of large numbers and the central limit theorem are asymptotic; learning needs non-asymptotic concentration inequalities.
- Markov $\Rightarrow$ Chebyshev (variance, $1/\delta$) and Chernoff (exponential moments) $\Rightarrow$ Hoeffding (bounded variables, $\ln(1/\delta)$), Bernstein (small variance, fast rates) and the kl bound. McDiarmid extends Hoeffding to functions with bounded differences.
- For variables in $[0, 1]$: with probability at least $1 - \delta$, $|\bar{X}_m - \mu| < \sqrt{\ln(2/\delta)/(2m)}$; with the union bound over K hypotheses, $\ln(2/\delta)$ becomes $\ln(2K/\delta)$.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

Expectation, variance and conditional expectation

- Q1.1.** True or false? If X and Y are two random variables with $\mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y]$, then X and Y are independent.
- Q1.2.** A classifier with error rate $p = 0.2$ is evaluated on $m = 400$ independent test examples. What are the expectation and the standard deviation of its test error?
- Q1.3.** Why can a density take values larger than 1, whereas a probability mass function cannot?
- Q1.4.** Use Jensen's inequality to compare $\mathbb{E}[\sqrt{Z}]$ and $\sqrt{\mathbb{E}[Z]}$ for a nonnegative variable Z , and $\mathbb{E}[e^X]$ and $e^{\mathbb{E}[X]}$.
- Q1.5.** Is the training error $\mathcal{R}_S(h_S)$ of the predictor h_S learned on S an unbiased estimator of its true risk? Is the error on an independent test set?

Limit theorems: law of large numbers and central limit theorem

- Q1.6.** What is the difference between the information given by the law of large numbers and by the central limit theorem?
- Q1.7.** Why does the Gaussian approximation work badly for the test error of a classifier with error rate $p = 0.01$ evaluated on $m = 100$ examples?

Q1.8. True or false? By the central limit theorem, for every $m \geq 30$, $\mathbb{P}(|\bar{X}_m - \mu| \geq 1.96\sigma/\sqrt{m}) \leq 0.05$.

Synthesis of the section

Q1.9. Why is Markov's inequality applied to $e^{\lambda X}$ (Chernoff's method) much stronger than when it is applied to X itself?

Q1.10. With $m = 500$ test examples, what is the half-width of the two-sided Hoeffding interval for the error rate of a fixed classifier at confidence 95%?

Q1.11. How does the sample size given by Hoeffding's inequality change if we want twice the precision ($\varepsilon/2$)? If we want $\delta/10$ instead of δ ?

Q1.12. True or false? Hoeffding's inequality requires the variables $X_1, \dots, X_m$ to be identically distributed.

Q1.13. Why does one need a union bound (or another uniform argument) to bound the true risk of the hypothesis h_S returned by empirical risk minimization?

Q1.14. In the uniform bound over a finite class, what happens to the half-width when the number of hypotheses goes from 10^3 to 10^6 , for $m = 10\,000$ and $\delta = 0.05$?

Q1.15. When is Bernstein's inequality much better than Hoeffding's inequality? Give an example from classification.

Q1.16. What are the bounded differences constants c_i of the function $f(x_1, \dots, x_m) = \max_i x_i$ for $x_i \in [0, 1]$, and what does McDiarmid's inequality give?

2 Linear Algebra and Convexity

This first section briefly introduces the mathematical tools required for the rest of the course¹. Throughout this section, we work in a real vector space E of finite dimension d , which can be identified with $\mathbb{R}^d$.

About this section

Guiding question. Which geometric and analytic tools do we need to write a learning problem as an optimization problem, and to know when it is “easy”?

In this section you will learn to:

- measure the size of vectors and matrices with norms, and interpret the inner product geometrically (angles, projections);
- compute gradients and Hessian matrices of functions of several variables;
- recognize convex sets and convex functions, and prove convexity from the definition or with the derivatives;
- read the eigenvalues of a symmetric matrix on the level sets of the associated quadratic form.

In a nutshell Almost every learning algorithm of this course minimizes a function of a vector of parameters (the weights of a linear model, of a neural network, etc.). Linear algebra gives us the language to describe these parameters and the geometry of the space they live in, while convexity tells us when this minimization is well-behaved: for a convex function, every local minimum is a global one.

2.1 Norms

Given two vectors $\mathbf{x}$ and $\mathbf{y}$ of $\mathbb{R}^d$, we define their inner product as the sum of the products of their components, *i.e.*

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_{j=1}^d x_j y_j.$$

When we compute the inner product of $\mathbf{x}$ with itself, we obtain the squared norm of $\mathbf{x}$:

$$\|\mathbf{x}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle.$$

Definition 2.1: Norm

Let E be a real vector space of dimension d . A map $\|\cdot\| : E \rightarrow \mathbb{R}$ is said to be a **norm** if, for all $\mathbf{u}, \mathbf{v} \in E$ and for all $\lambda \in \mathbb{R}$, the following properties hold:

- $\|\mathbf{u}\| \geq 0$ (nonnegativity),

¹You can also read the Linear Algebra course that I wrote for second-year Bachelor students, which is available on my website.

- $\|\mathbf{u}\| = 0 \iff \mathbf{u} = \mathbf{0}$ (definiteness),
- $\|\lambda\mathbf{u}\| = |\lambda| \|\mathbf{u}\|$ (absolute homogeneity),
- $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ (triangle inequality).

A norm can be used to measure the distance between two vectors $\mathbf{x}, \mathbf{y}$ of the same vector space:

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|.$$

The following norms are the most commonly used:

- the Manhattan norm, also known as the L_1 -norm:

$$\|\mathbf{x}\|_1 = \sum_{j=1}^d |x_j|;$$

- the Euclidean norm, also known as the L_2 -norm:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{j=1}^d x_j^2};$$

- the L_∞ -norm (or max norm):

$$\|\mathbf{x}\|_\infty = \max_{j \in [1, d]} |x_j|;$$

- more generally, for any real number $p \geq 1$, the L_p -norm:

$$\|\mathbf{x}\|_p = \left(\sum_{j=1}^d |x_j|^p \right)^{1/p}.$$

It is also possible to define more complex norms when we work with objects other than vectors, such as matrices or functions.

Example 2.1. Let us show that the Euclidean norm is indeed a norm. We have to check each property of the definition. Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ and $\lambda \in \mathbb{R}$.

- It is obvious that $\|\mathbf{x}\|_2 = \sqrt{\sum_{j=1}^d |x_j|^2}$ is nonnegative.
- Each $|x_j|^2$ is a nonnegative number, and a sum of nonnegative numbers is equal to zero if and only if all of them are equal to zero. Hence $\|\mathbf{x}\|_2 = 0 \Rightarrow \mathbf{x} = \mathbf{0}$, and the converse is obvious.
- We then focus on absolute homogeneity:

$$\begin{aligned}
\|\lambda \mathbf{x}\|_2 &= \sqrt{\sum_{j=1}^d (\lambda x_j)^2} \\
&= \sqrt{\sum_{j=1}^d |\lambda|^2 x_j^2} \\
&\quad \downarrow \lambda \text{ does not depend on the index } j \\
&= |\lambda| \sqrt{\sum_{j=1}^d x_j^2}.
\end{aligned}$$

- To prove the triangle inequality, we use the **Cauchy–Schwarz inequality**, which holds for any inner product and the norm it induces (here, the Euclidean inner product and the L_2 -norm):

$$|\langle \mathbf{x}, \mathbf{y} \rangle| \leq \|\mathbf{x}\|_2 \|\mathbf{y}\|_2.$$

Thus,

$$\begin{aligned}
\|\mathbf{x} + \mathbf{y}\|_2^2 &= \|\mathbf{x}\|_2^2 + 2\langle \mathbf{x}, \mathbf{y} \rangle + \|\mathbf{y}\|_2^2 \\
&\quad \downarrow \text{using the Cauchy–Schwarz inequality} \\
&\leq \|\mathbf{x}\|_2^2 + 2\|\mathbf{x}\|_2 \|\mathbf{y}\|_2 + \|\mathbf{y}\|_2^2 \\
&= (\|\mathbf{x}\|_2 + \|\mathbf{y}\|_2)^2.
\end{aligned}$$

Taking the square root, which is an increasing function, gives the result.

Geometric meaning of the inner product The Cauchy–Schwarz inequality states that, for nonzero vectors, the ratio $\langle \mathbf{x}, \mathbf{y} \rangle / (\|\mathbf{x}\|_2 \|\mathbf{y}\|_2)$ lies in $[-1, 1]$. It is therefore the cosine of an angle $\theta \in [0, \pi]$, the **angle** between $\mathbf{x}$ and $\mathbf{y}$:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta.$$

In particular, $\mathbf{x}$ and $\mathbf{y}$ are **orthogonal** when $\langle \mathbf{x}, \mathbf{y} \rangle = 0$. The inner product also gives the **orthogonal projection** of $\mathbf{x}$ onto the line spanned by a nonzero vector $\mathbf{y}$:

$$\mathbf{p} = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{y}\|_2^2} \mathbf{y}, \quad \text{with} \quad \langle \mathbf{x} - \mathbf{p}, \mathbf{y} \rangle = 0,$$

see Figure 8. When $\|\mathbf{y}\|_2 = 1$, the length of the projection is simply $|\langle \mathbf{x}, \mathbf{y} \rangle|$: *the inner product with a unit vector measures how far $\mathbf{x}$ goes in the direction $\mathbf{y}$.*

Remark In Machine Learning, the quantity $\cos \theta = \langle \mathbf{x}, \mathbf{y} \rangle / (\|\mathbf{x}\|_2 \|\mathbf{y}\|_2)$ is known as the *cosine similarity*; it is widely used to compare documents or embeddings. Similarly, a linear model predicts with $\langle \mathbf{w}, \mathbf{x} \rangle + b$: the sign of the prediction tells

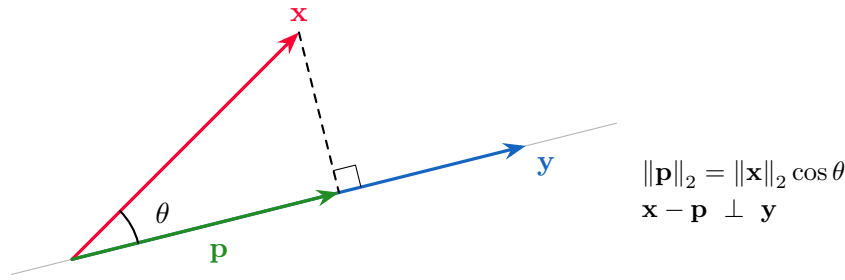


Figure 8: Geometric view of the inner product: $\langle \mathbf{x}, \mathbf{y} \rangle = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta$, and $\mathbf{p} = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{y}\|_2^2} \mathbf{y}$ is the orthogonal projection of $\mathbf{x}$ onto the line spanned by $\mathbf{y}$ (here θ is acute, so $\langle \mathbf{x}, \mathbf{y} \rangle > 0$).

on which side of the hyperplane $\{\mathbf{x} \mid \langle \mathbf{w}, \mathbf{x} \rangle + b = 0\}$ the point $\mathbf{x}$ lies, and, when $\|\mathbf{w}\|_2 = 1$, its absolute value is the distance from $\mathbf{x}$ to this hyperplane.

We can also represent the unit balls associated with the norms defined above, see Figure 9. For every $p \geq 1$, the unit ball is a convex set, which reflects the fact that every norm is a convex function (for $p < 1$, the quantity $\|\cdot\|_p$ is no longer a norm and its unit ball is not convex). Figure 10 shows how the unit ball deforms continuously from the diamond ($p = 1$) to the square ($p = \infty$), and what goes wrong when $p < 1$: the triangle inequality fails and the ball has “inward” corners.

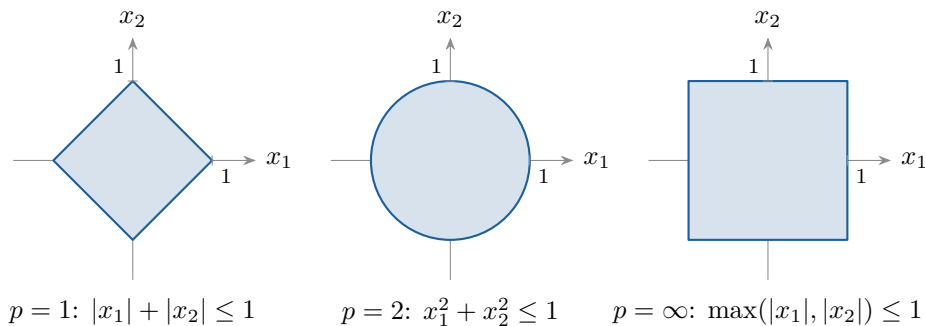


Figure 9: Unit balls $\{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_p \leq 1\}$ of the L_1 -norm (a “diamond”), of the L_2 -norm (a disk) and of the L_∞ -norm (a square). All three are convex, and they are nested: diamond $\subset$ disk $\subset$ square.

In a nutshell All norms on $\mathbb{R}^d$ are equivalent (for instance $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{d} \|\mathbf{x}\|_2$), so they define the same notion of convergence; but they do not have the same shape. This shape matters in Machine Learning: when the norm of the parameters is used as a penalty (regularization), the corners of the L_1 ball on the axes favour solutions with many zero coordinates (sparse models, the Lasso), whereas the round L_2 ball only shrinks the coordinates (ridge regression).

Let us now focus on norms on matrices, and more specifically on the Frobenius norm.

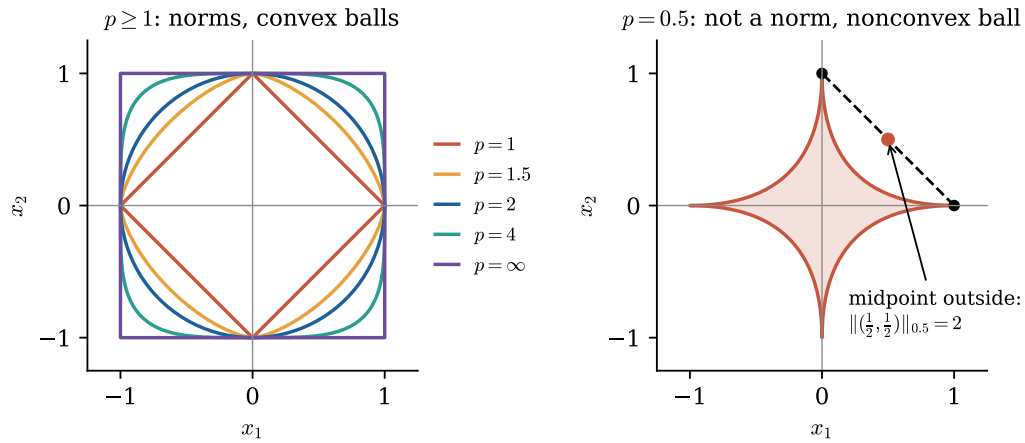


Figure 10: Left: unit spheres $\{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_p = 1\}$ for several $p \geq 1$; the balls are nested, $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1$, and all of them are convex. Right: for $p = 0.5$, the points $(1, 0)$ and $(0, 1)$ belong to the “ball” but their midpoint does not, so the set is not convex and $\|\cdot\|_{0.5}$ is not a norm.

[code: make_figures_ch01.py: fig_ch01_lp_balls]

Definition 2.2: Frobenius norm on Matrices

Given two matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m \times d}$, their inner product is defined by

$$\langle \mathbf{X}, \mathbf{Y} \rangle = \text{Tr}(\mathbf{X}^T \mathbf{Y}) = \sum_{i=1}^m \sum_{j=1}^d X_{ij} Y_{ij}.$$

The norm induced by this inner product is the **Frobenius norm**:

$$\|\mathbf{X}\|_F = \sqrt{\text{Tr}(\mathbf{X}^T \mathbf{X})} = \left(\sum_{i=1}^m \sum_{j=1}^d X_{ij}^2 \right)^{1/2}.$$

2.2 Derivatives

The notion of derivative, or more generally of gradient, is fundamental in optimization. It is at the heart of gradient descent algorithms, and it will also allow us to characterize convex functions. As we will see, this characterization can be made by studying the first-order derivative or, often more simply, the second-order derivative.

A few reminders Let us start with a reminder on the derivative of a function of one variable.

Definition 2.3: Derivative

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function and $x_0 \in \mathbb{R}$. We say that f is **differentiable** at x_0 if the limit

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

exists and is finite.

Equivalently, f is differentiable at x_0 if, for h close to 0, we have

$$f(x_0 + h) = f(x_0) + hf'(x_0) + h\varepsilon(h),$$

where $\varepsilon(h)$ is a function that tends to 0 when h tends to 0.

The last formula is called the first-order approximation of the function f , and it can be generalized to any order. This is what we call the Taylor expansion of a function.

Proposition 2.1: Taylor's Theorem

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function that is k times differentiable, with $k \geq 1$, at a point x_0 . Then there exists a function $\varepsilon_k : \mathbb{R} \rightarrow \mathbb{R}$ such that

$$f(x_0 + h) = f(x_0) + hf'(x_0) + f''(x_0)\frac{h^2}{2} + \dots + f^{(k)}(x_0)\frac{h^k}{k!} + h^k\varepsilon_k(h),$$

where $\lim_{h \rightarrow 0} \varepsilon_k(h) = 0$.

First- and second-order derivatives of a function of several variables In Machine Learning, as the reader may have seen with linear models, the number of parameters to learn is almost always greater than 1. We are therefore most often led to study real-valued functions of several variables, *i.e.* functions which take a vector as input and return a real value.

Definition 2.4: First-order Derivative

Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a continuous function and $\mathbf{x}_0 \in \mathbb{R}^d$. Then f is **differentiable** at $\mathbf{x}_0$ if there exists a (row) vector $Jf(\mathbf{x}_0) \in \mathbb{R}^{1 \times d}$ such that

$$\lim_{\mathbf{x} \rightarrow \mathbf{x}_0} \frac{|f(\mathbf{x}) - f(\mathbf{x}_0) - Jf(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)|}{\|\mathbf{x} - \mathbf{x}_0\|} = 0.$$

For all i , the components of $Jf(\mathbf{x}_0)$, called the **Jacobian** of f at $\mathbf{x}_0$, are given by the partial derivatives

$$J_i f(\mathbf{x}_0) = \left. \frac{\partial f(\mathbf{x})}{\partial x_i} \right|_{\mathbf{x}=\mathbf{x}_0}.$$

The column vector $\nabla f(\mathbf{x}_0) = Jf(\mathbf{x}_0)^T$ is called the **gradient** of f at $\mathbf{x}_0$.

The gradient makes it possible to approximate the function in the neighborhood of the point where it is computed, *i.e.* in a neighborhood $V(\mathbf{x}_0)$ of $\mathbf{x}_0$. For all $\mathbf{x} \in V(\mathbf{x}_0)$, we have

$$f(\mathbf{x}) \simeq f(\mathbf{x}_0) + \langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle.$$

It is this approximation that will provide a first characterization of convex functions. In the following, we often identify the Jacobian with the gradient, and we write $\nabla f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$ or $\langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$ indifferently.

In a nutshell Moving from $\mathbf{x}_0$ by a small step $h\mathbf{d}$ in a unit direction $\mathbf{d}$ changes the value of f by approximately $h\langle \nabla f(\mathbf{x}_0), \mathbf{d} \rangle$. By the Cauchy–Schwarz inequality, this quantity is maximal when $\mathbf{d} = \nabla f(\mathbf{x}_0) / \|\nabla f(\mathbf{x}_0)\|_2$ and minimal for the opposite direction: *the gradient points in the direction of steepest ascent*, and $-\nabla f(\mathbf{x}_0)$ in the direction of steepest descent. This simple observation is the starting point of the gradient descent algorithm of Section 3. In the same way, the gradient is orthogonal to the level set $\{\mathbf{x} \mid f(\mathbf{x}) = f(\mathbf{x}_0)\}$, since f does not change to first order along this set.

Example 2.2. Let us consider the function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ defined by

$$f(x_1, x_2, x_3) = 3x_1^2 + 2x_1x_2x_3 + 6x_3 + 5x_2x_3 + 9x_1x_3.$$

We want to compute the Jacobian (or gradient) of this function. To do so, we need to compute

$$\frac{\partial f}{\partial x_1}, \quad \frac{\partial f}{\partial x_2} \quad \text{and} \quad \frac{\partial f}{\partial x_3}.$$

The Jacobian of f at (x_1, x_2, x_3) is given by

$$Jf(x_1, x_2, x_3) = \begin{pmatrix} 6x_1 + 2x_2x_3 + 9x_3 & 2x_1x_3 + 5x_3 & 2x_1x_2 + 6 + 5x_2 + 9x_1 \end{pmatrix}.$$

Example 2.3. We consider the log-sum-exp function, defined for all $\mathbf{x} \in \mathbb{R}^d$ and a fixed vector $\mathbf{b} \in \mathbb{R}^d$ by

$$f(\mathbf{x}, \mathbf{b}) = \ln \sum_{j=1}^d \exp(x_j + b_j).$$

When $\mathbf{b} = \mathbf{0}$, this function is known to be a good (and smooth!) approximation of the function $\mathbf{x} \mapsto \max_j x_j$.

The gradient $\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{b})$ of this function (written here as a row vector) is given by

$$\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{b}) = \left(\frac{\exp(x_1 + b_1)}{\sum_{j=1}^d \exp(x_j + b_j)} \quad \cdots \quad \frac{\exp(x_d + b_d)}{\sum_{j=1}^d \exp(x_j + b_j)} \right).$$

Its components are positive and sum to 1: this is the *softmax* function, which we will meet again with neural networks.

Definition 2.5: Second-order Derivative

Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a real-valued function. Provided that this function is twice differentiable at $\mathbf{x}_0$, its second-order derivative at $\mathbf{x}_0$, also called the **Hessian** of f at $\mathbf{x}_0$, is the matrix $\mathbf{H} = \nabla_{\mathbf{x}}^2 f(\mathbf{x}_0) \in \mathbb{R}^{d \times d}$ whose entries are given by

$$H_{ij} = \left. \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \right|_{\mathbf{x}=\mathbf{x}_0}.$$

This matrix is useful to prove whether a function f is convex, and also to build efficient algorithms.

Example 2.4. Let us consider the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ defined by

$$f(x_1, x_2) = 4x_1^2 + 6x_2^2 + 3x_1x_2 + 2(\cos(x_1) + \sin(x_2)),$$

for which we aim to compute the Hessian matrix. We first have to compute the Jacobian of this function:

$$Jf(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 8x_1 + 3x_2 - 2\sin(x_1) & 12x_2 + 3x_1 + 2\cos(x_2) \end{pmatrix}.$$

The Hessian matrix is then given by

$$\mathbf{H}_f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{pmatrix} = \begin{pmatrix} 8 - 2\cos(x_1) & 3 \\ 3 & 12 - 2\sin(x_2) \end{pmatrix}.$$

2.3 Convex sets

The notion of convex set is important when we want to build optimization algorithms and guarantee that the successive iterates of these algorithms remain in a given set. Moreover, it will also allow us to characterize convex functions.

Let us start with the definition of a convex set.

Definition 2.6: Convex Set

A set C is said to be **convex** if, for all $\mathbf{u}, \mathbf{v} \in C$ and for all $t \in [0, 1]$, we have

$$t\mathbf{u} + (1 - t)\mathbf{v} \in C.$$

In other words, C is convex if **every point of the segment joining $\mathbf{u}$ and $\mathbf{v}$ belongs to the set.**

Figure 11 provides some examples of convex and nonconvex sets. The first and third shapes represent nonconvex sets: for each of them, we can find two points of the

set such that the segment joining them leaves the set. For instance, for the ring, if we take two diametrically opposed points, the segment joining them goes through the hole in the center, which does not belong to the set. The second and fourth shapes represent convex sets: whichever two points we pick, the segment joining them stays inside.

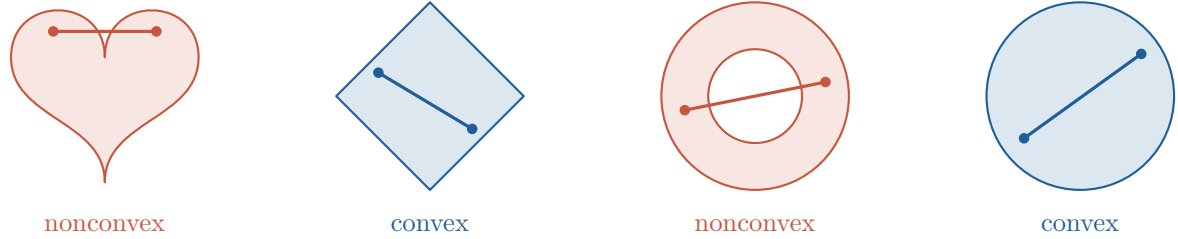


Figure 11: Two nonconvex sets (red) and two convex sets (blue). For the nonconvex sets, the segment joining the two marked points leaves the set (it crosses the notch of the heart, and the hole of the ring); for the convex sets, every such segment stays inside.

Example 2.5. Here are some fundamental examples of convex sets:

- the unit ball $\mathcal{B} = \{\mathbf{u} \in \mathbb{R}^d \mid \|\mathbf{u}\| \leq 1\}$ of any norm;
- every interval of $\mathbb{R}$;
- every hyperplane $\{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{a}^T \mathbf{x} = b\}$ and every half-space $\{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{a}^T \mathbf{x} \leq b\}$;
- if C_1 and C_2 are two convex sets, then their intersection $C = C_1 \cap C_2$ is also convex.

Let us study the unit ball, which we have claimed above to be convex, and prove it for the L_2 -norm. Consider $\lambda \in [0, 1]$ and two vectors $\mathbf{u}, \mathbf{v}$ of the unit ball, and let us set $\mathbf{z} = \lambda \mathbf{u} + (1 - \lambda) \mathbf{v}$. Then

$$\begin{aligned}
 \|\mathbf{z}\|_2 &= \|\lambda \mathbf{u} + (1 - \lambda) \mathbf{v}\|_2 \\
 &\quad \downarrow \text{apply the triangle inequality} \\
 &\leq \|\lambda \mathbf{u}\|_2 + \|(1 - \lambda) \mathbf{v}\|_2 \\
 &\quad \downarrow \text{absolute homogeneity of the norm, with } \lambda \geq 0 \text{ and } 1 - \lambda \geq 0 \\
 &= \lambda \|\mathbf{u}\|_2 + (1 - \lambda) \|\mathbf{v}\|_2 \\
 &\quad \downarrow \text{both } \mathbf{u} \text{ and } \mathbf{v} \text{ belong to the unit ball} \\
 &\leq \lambda + (1 - \lambda) = 1.
 \end{aligned}$$

Hence $\mathbf{z}$ belongs to the unit ball.

Proposition 2.2: Convex combination

Let $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m$ be m points of a convex set C . Then, for all nonnegative numbers $\lambda_1, \lambda_2, \dots, \lambda_m$ such that $\sum_{i=1}^m \lambda_i = 1$,

$$\mathbf{v} = \sum_{i=1}^m \lambda_i \mathbf{u}_i \in C.$$

In other words, every convex combination of points of a convex set belongs to this convex set.

Given a finite set of points $\mathbf{x}_1, \dots, \mathbf{x}_m$, the convex combinations of these points form a convex set. This set, called the **convex hull** $\mathcal{H}$ of the points, is the smallest convex set that contains all of them:

$$\mathcal{H} = \left\{ \mathbf{x} = \sum_{i=1}^m \lambda_i \mathbf{x}_i \mid \lambda_1, \dots, \lambda_m \geq 0, \sum_{i=1}^m \lambda_i = 1 \right\},$$

see Figure 12.

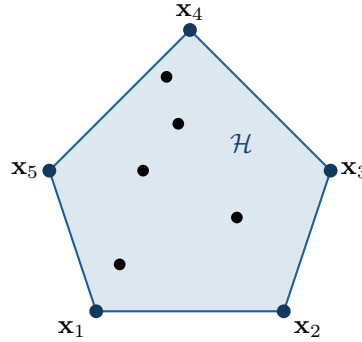


Figure 12: Convex hull $\mathcal{H}$ of ten points of the plane: it is the convex polygon whose vertices $\mathbf{x}_1, \dots, \mathbf{x}_5$ are five of the points; the five other points lie inside $\mathcal{H}$ and do not change it.

2.4 Convex functions

The study of convex functions is essential in Machine Learning. Indeed, we will see that most of the problems we will try to solve involve convex functions. Moreover, it is through the study of this type of problem that we will be able to provide theoretical guarantees on the performance of our algorithms.

Fundamental definitions Let us start with the definition of a convex function.

Definition 2.7: Convex functions

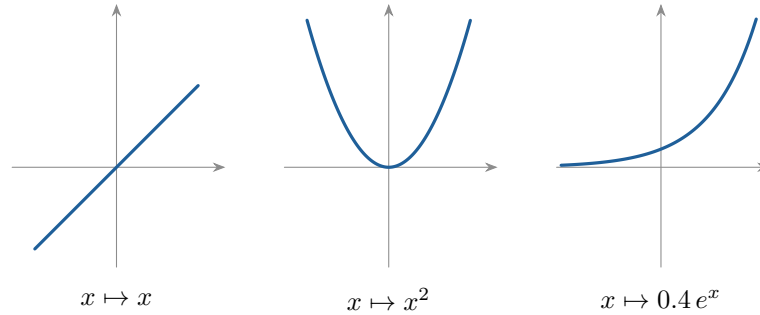
Let $\mathcal{U}$ be a nonempty convex subset of a real vector space of dimension d (i.e. $\mathcal{U} \subset \mathbb{R}^d$). A function $f : \mathcal{U} \rightarrow \mathbb{R}$ is said to be **convex** if, for all $\mathbf{u}, \mathbf{v} \in \mathcal{U}$ and for all $t \in [0, 1]$, we have

$$f(t\mathbf{u} + (1-t)\mathbf{v}) \leq tf(\mathbf{u}) + (1-t)f(\mathbf{v}).$$

Example 2.6. The following functions are convex:

- any linear (or affine) function,
- $f : \mathbb{R} \rightarrow \mathbb{R}, \quad f(x) = x^2,$

- $f : \mathbb{R} \rightarrow \mathbb{R}, \quad f(x) = 0.4 \times \exp(x).$



Proposition 2.3: Convexity and Restriction

A function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is convex if and only if its restriction to any line is convex, *i.e.* if and only if the function $g : t \mapsto f(\mathbf{x} + t\mathbf{v})$ is convex for all $\mathbf{x}$ and $\mathbf{v}$ (on the set of $t \in \mathbb{R}$ such that $\mathbf{x} + t\mathbf{v}$ belongs to the domain of f). Similarly, f is concave if and only if all such functions g are concave.

Proof. It only requires writing down the definition of a convex function. Assume that f is convex and let t_1, t_2 be two real numbers and $\lambda \in [0, 1]$. Then

$$g(\lambda t_1 + (1 - \lambda)t_2) = f(\lambda(\mathbf{x} + t_1\mathbf{v}) + (1 - \lambda)(\mathbf{x} + t_2\mathbf{v})) \leq \lambda g(t_1) + (1 - \lambda)g(t_2),$$

so g is convex. Conversely, let $\mathbf{u}, \mathbf{v}$ be two points of the domain of f and $t \in [0, 1]$, and consider the restriction $g(t) = f(\mathbf{v} + t(\mathbf{u} - \mathbf{v}))$ of f to the line going through $\mathbf{v}$ and $\mathbf{u}$. If g is convex, then

$$f(t\mathbf{u} + (1 - t)\mathbf{v}) = g(t) = g(t \cdot 1 + (1 - t) \cdot 0) \leq tg(1) + (1 - t)g(0) = tf(\mathbf{u}) + (1 - t)f(\mathbf{v}),$$

which proves that f is convex. The concave case follows by applying this result to $-f$. $\square$

Figure 13 provides an illustration of the definition of convex functions, and Figure 14 contrasts it with a nonconvex function.

Remark (Jensen's inequality) By induction on the number of points, the definition of convexity extends to any convex combination: if f is convex, then for all $\mathbf{u}_1, \dots, \mathbf{u}_m \in \mathcal{U}$ and all nonnegative $\lambda_1, \dots, \lambda_m$ summing to 1,

$$f\left(\sum_{i=1}^m \lambda_i \mathbf{u}_i\right) \leq \sum_{i=1}^m \lambda_i f(\mathbf{u}_i).$$

Reading the weights λ_i as the probabilities of a random variable U taking the values $\mathbf{u}_i$, this is $f(\mathbb{E}[U]) \leq \mathbb{E}[f(U)]$, the probabilistic form of Jensen's inequality, which will be used repeatedly in the statistical parts of the course (*e.g.* in Part IV).

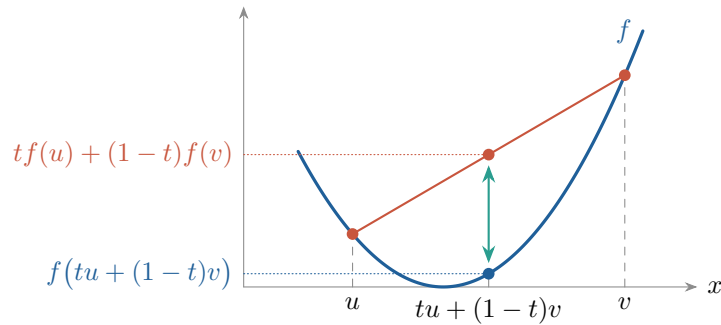


Figure 13: Illustration of the definition of a convex function f (here $f(x) = x^2$, $u = -1$, $v = 2$, $t = 1/2$): the chord (red) joining $(u, f(u))$ and $(v, f(v))$ lies above the graph, *i.e.* $f(tu + (1-t)v) \leq tf(u) + (1-t)f(v)$ for every $t \in [0, 1]$; the teal arrow shows the nonnegative gap between the two sides.

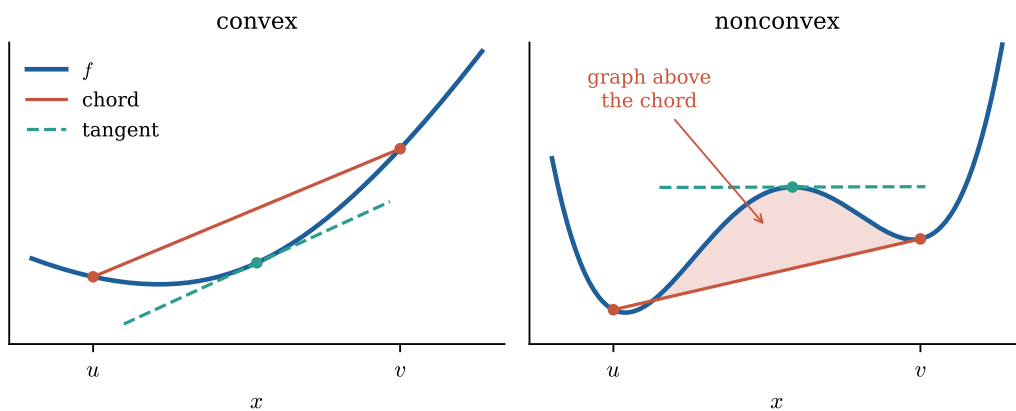


Figure 14: Left: for a convex function, every chord lies above the graph and every tangent lies below it. Right: for a nonconvex function (a “double well”), the graph goes above the chord joining u and v (shaded area), and the tangent at the local maximum goes above the graph. Note that the nonconvex function has two local minima, only one of which is global. [code: `make_figures_ch01.py: fig_ch01_convex_vs_nonconvex`]

Historical note Convex functions were introduced and studied systematically by the Danish mathematician Johan Jensen, who proved the inequality that bears his name for finite convex combinations (Jensen, 1906). Jensen worked as an engineer for the Copenhagen telephone company and did mathematics in his spare time. A modern reference on convexity and optimization, freely available online, is the book of Boyd and Vandenberghe (2004).

The next definition makes it possible to link convex functions with convex sets.

Definition 2.8: Epigraph

Let $\mathcal{U}$ be a nonempty subset of a real vector space of dimension d (*i.e.* $\mathcal{U} \subset \mathbb{R}^d$) and let $f : \mathcal{U} \rightarrow \mathbb{R}$ be a function. The **epigraph** of f , denoted by E_f , is the set of points lying above the graph of f , *i.e.*

$$E_f = \{(\mathbf{x}, y) \in \mathcal{U} \times \mathbb{R} \mid f(\mathbf{x}) \leq y\}.$$

Proposition 2.4: Convexity via Epigraph

Let $\mathcal{U}$ be a nonempty convex subset of a real vector space of dimension d (i.e. $\mathcal{U} \subset \mathbb{R}^d$). A function f is convex on $\mathcal{U}$ if and only if its epigraph E_f is a convex set.

Proof. It is again a direct application of the definitions. Assume that f is convex, and let $(\mathbf{x}_1, y_1)$ and $(\mathbf{x}_2, y_2)$ be two points of E_f and $t \in [0, 1]$. Then

$$f(t\mathbf{x}_1 + (1-t)\mathbf{x}_2) \leq tf(\mathbf{x}_1) + (1-t)f(\mathbf{x}_2) \leq ty_1 + (1-t)y_2,$$

i.e. $t(\mathbf{x}_1, y_1) + (1-t)(\mathbf{x}_2, y_2) \in E_f$: the epigraph is convex. Conversely, assume that E_f is convex and let $\mathbf{u}, \mathbf{v} \in \mathcal{U}$ and $t \in [0, 1]$. Since $(\mathbf{u}, f(\mathbf{u}))$ and $(\mathbf{v}, f(\mathbf{v}))$ belong to E_f , so does the point $(t\mathbf{u} + (1-t)\mathbf{v}, tf(\mathbf{u}) + (1-t)f(\mathbf{v}))$, which exactly means that $f(t\mathbf{u} + (1-t)\mathbf{v}) \leq tf(\mathbf{u}) + (1-t)f(\mathbf{v})$. $\square$

This proposition is illustrated in Figure 15.

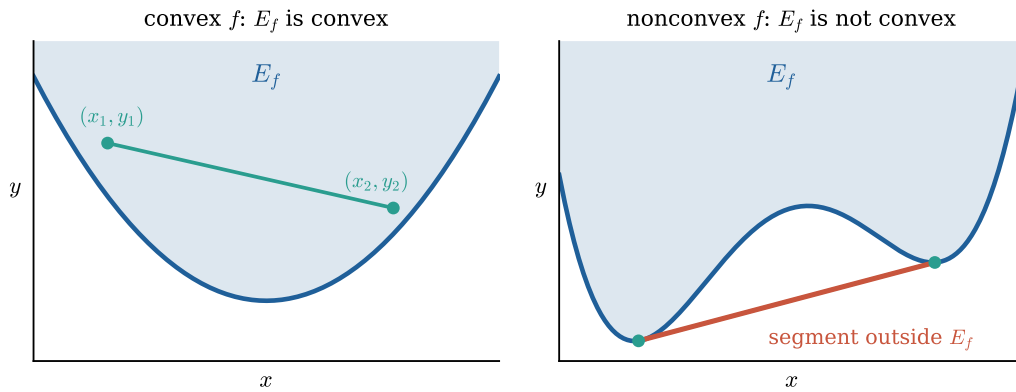


Figure 15: Left: the epigraph E_f (shaded) of a convex function is a convex set: the segment joining two of its points stays inside. Right: for a nonconvex function, the segment joining two points of the graph goes below the graph, i.e. it leaves the epigraph, which is therefore not convex.

[code: make_figures_ch01.py: fig_ch01_epigraph]

Definition 2.9: Concave functions

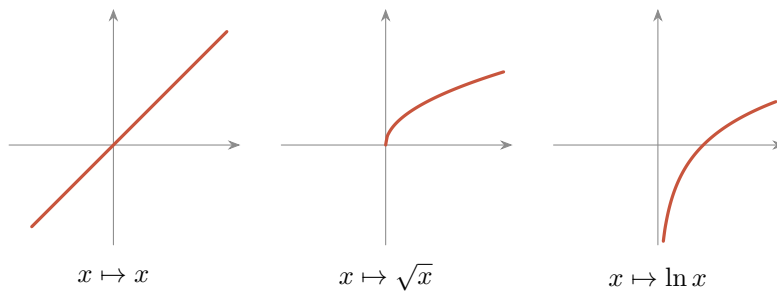
Let $\mathcal{U}$ be a nonempty convex subset of a real vector space of dimension d (i.e. $\mathcal{U} \subset \mathbb{R}^d$). A function $f: \mathcal{U} \rightarrow \mathbb{R}$ is said to be **concave** if, for all $\mathbf{u}, \mathbf{v} \in \mathcal{U}$ and for all $t \in [0, 1]$, we have

$$f(t\mathbf{u} + (1-t)\mathbf{v}) \geq tf(\mathbf{u}) + (1-t)f(\mathbf{v}).$$

With this definition, we directly see that f is concave if and only if $-f$ is convex. Furthermore, it is also possible to characterize concave functions through the convexity of their hypograph, *i.e.* the region below the graph of the function.

Example 2.7. The following functions are concave:

- any linear (or affine) function,
- $f : \mathbb{R}_+ \rightarrow \mathbb{R}, \quad f(x) = \sqrt{x},$
- $f : \mathbb{R}_+^* \rightarrow \mathbb{R}, \quad f(x) = \ln(x).$



The following proposition provides some properties of convex functions.

Proposition 2.5: Properties of Convex Functions

Let f and g be two convex functions defined on a convex subset $\mathcal{U}$ of $\mathbb{R}^d$. Then:

- any linear combination of f and g with nonnegative coefficients, $\alpha f + \beta g$ with $\alpha, \beta \geq 0$, is convex (in particular, any convex combination of f and g);
- if $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ is convex and **nondecreasing**, then $\varphi \circ g$ is convex;
- the function h defined by $h(\mathbf{x}) = \max(f(\mathbf{x}), g(\mathbf{x}))$ is convex.

Proof. The proof of this proposition only relies on the definition of convexity; it is left to the reader as an exercise. $\square$

Pitfall The assumption that φ is **nondecreasing** cannot be dropped. The function $\varphi(t) = e^{-t}$ is convex and $g(x) = x^2$ is convex, but $\varphi \circ g(x) = e^{-x^2}$ is the Gaussian bell, which is not convex (its second derivative $(4x^2 - 2)e^{-x^2}$ is negative for $|x| < 1/\sqrt{2}$). Similarly, the product of two convex functions or the minimum of two convex functions are not convex in general: think of $x \mapsto x \cdot x^2 = x^3$ or of $x \mapsto \min((x-1)^2, (x+1)^2)$.

Remark Another very useful rule: *the composition of a convex function with an affine map is convex*. If $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is convex, $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $\mathbf{b} \in \mathbb{R}^n$, then $\mathbf{x} \mapsto f(\mathbf{Ax} + \mathbf{b})$ is convex on $\mathbb{R}^d$. This is why the losses used in Machine Learning (squared loss, hinge loss, logistic loss, *etc.*) are convex *with respect to*

the parameters of a linear model: the prediction $\langle \mathbf{w}, \mathbf{x} \rangle + b$ is affine in $(\mathbf{w}, b)$.

Characterization of convexity using derivatives In the following, we study how the convexity of a function f can be characterized using its first- and second-order derivatives.

Proposition 2.6: First-order derivative and convexity

Let f be a continuously differentiable function on an open convex set $\mathcal{U} \subset \mathbb{R}^d$. Then f is convex if and only if, for all $\mathbf{u}, \mathbf{v} \in \mathcal{U}$, we have

$$f(\mathbf{v}) \geq f(\mathbf{u}) + \langle \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle.$$

Equivalently, f is convex if and only if, for all $\mathbf{u}, \mathbf{v} \in \mathcal{U}$, we have

$$\langle \nabla f(\mathbf{v}) - \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle \geq 0.$$

Proof. Convexity $\Rightarrow$ first inequality. Let $\mathbf{u}, \mathbf{v} \in \mathcal{U}$ and $t \in]0, 1]$. By convexity, $f(\mathbf{u} + t(\mathbf{v} - \mathbf{u})) \leq f(\mathbf{u}) + t(f(\mathbf{v}) - f(\mathbf{u}))$, hence

$$\frac{f(\mathbf{u} + t(\mathbf{v} - \mathbf{u})) - f(\mathbf{u})}{t} \leq f(\mathbf{v}) - f(\mathbf{u}).$$

Letting $t \rightarrow 0$, the left-hand side tends to $\langle \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle$, which gives the first inequality.

First inequality $\Rightarrow$ convexity. Let $\mathbf{u}, \mathbf{v} \in \mathcal{U}$, $t \in [0, 1]$ and $\mathbf{w} = t\mathbf{u} + (1 - t)\mathbf{v}$. Applying the inequality at the point $\mathbf{w}$ gives $f(\mathbf{u}) \geq f(\mathbf{w}) + \langle \nabla f(\mathbf{w}), \mathbf{u} - \mathbf{w} \rangle$ and $f(\mathbf{v}) \geq f(\mathbf{w}) + \langle \nabla f(\mathbf{w}), \mathbf{v} - \mathbf{w} \rangle$. Multiplying the first inequality by t , the second one by $1 - t$ and summing them, we get

$$tf(\mathbf{u}) + (1 - t)f(\mathbf{v}) \geq f(\mathbf{w}) + \langle \nabla f(\mathbf{w}), t\mathbf{u} + (1 - t)\mathbf{v} - \mathbf{w} \rangle = f(\mathbf{w}).$$

First inequality $\Rightarrow$ second inequality. Write the first inequality for the pair $(\mathbf{u}, \mathbf{v})$ and for the pair $(\mathbf{v}, \mathbf{u})$, and sum them.

Second inequality $\Rightarrow$ first inequality. Let $\varphi(t) = f(\mathbf{u} + t(\mathbf{v} - \mathbf{u}))$ for $t \in [0, 1]$. Then $\varphi'(t) = \langle \nabla f(\mathbf{u} + t(\mathbf{v} - \mathbf{u})), \mathbf{v} - \mathbf{u} \rangle$, and applying the second inequality to the points $\mathbf{u} + t(\mathbf{v} - \mathbf{u})$ and $\mathbf{u}$ (and dividing by $t > 0$) gives $\varphi'(t) \geq \langle \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle$. Hence

$$f(\mathbf{v}) - f(\mathbf{u}) = \varphi(1) - \varphi(0) = \int_0^1 \varphi'(t) dt \geq \langle \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle.$$

□

Proposition 2.6 is illustrated in Figure 16: the tangents of a convex function always lie below its graph. Note that, for a concave function, the tangents lie above the graph.

In a nutshell The first-order condition is remarkable: a *local* information, the value and the gradient of f at a single point $\mathbf{u}$, gives a *global* lower bound on f over the whole set $\mathcal{U}$. In particular, if $\nabla f(\mathbf{u}^*) = \mathbf{0}$, then $f(\mathbf{v}) \geq f(\mathbf{u}^*)$ for every $\mathbf{v} \in \mathcal{U}$: for a differentiable convex function, every critical point is a global minimum. This is the main reason why convex problems are considered “easy” in optimization, and why we try to formulate learning problems as convex problems whenever possible.

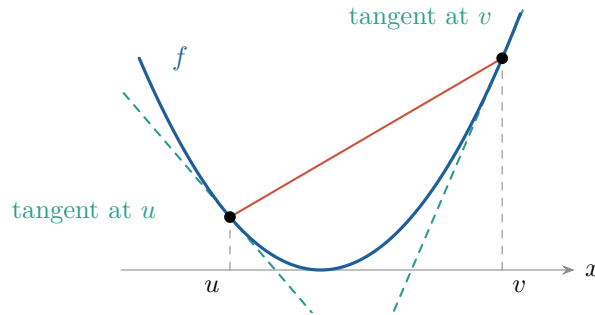


Figure 16: The convex function $f(x) = x^2$ with its tangents (dashed) at the points u and v , and the chord joining them (red). The tangents of a convex function always lie below its graph, while its chords lie above it.

Proposition 2.7: Second-order derivative and convexity

Let f be a twice continuously differentiable function on an open convex subset $\mathcal{U}$ of $\mathbb{R}^d$, and denote by $\nabla^2 f(\mathbf{u})$ its Hessian matrix at $\mathbf{u}$. Then f is convex if and only if its Hessian is positive semidefinite (PSD) at every point, which we write $\nabla^2 f(\mathbf{u}) \succeq 0$ for all $\mathbf{u} \in \mathcal{U}$, i.e.

$$\forall \mathbf{u} \in \mathcal{U}, \forall \mathbf{z} \in \mathbb{R}^d, \quad \mathbf{z}^T \nabla^2 f(\mathbf{u}) \mathbf{z} \geq 0.$$

Remember that a symmetric matrix $\mathbf{H}$ is PSD if and only if all its eigenvalues are nonnegative (this is a consequence of the spectral theorem recalled below). It is sometimes easier to check this property than to apply the definition of convexity to a complicated function.

Pitfall Positive semidefiniteness is **not** a property of the entries of the matrix.

The matrix $\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ has only positive entries, but its eigenvalues are 3 and -1 : with $\mathbf{z} = (1, -1)^T$, we get $\mathbf{z}^T \mathbf{H} \mathbf{z} = -2 < 0$. Conversely, a PSD matrix may have negative off-diagonal entries. Also note that a positive *definite* Hessian ($\nabla^2 f(\mathbf{u}) \succ 0$ everywhere) implies that f is *strictly* convex, but the converse is false: $x \mapsto x^4$ is strictly convex while its second derivative vanishes at 0.

A PSD Hessian means that, along every direction $\mathbf{z}$, the directional derivative $t \mapsto \langle \nabla f(\mathbf{u} + t\mathbf{z}), \mathbf{z} \rangle$ is a nondecreasing function of t . This is consistent with Proposition 2.6: a differentiable function is convex if and only if its gradient is “nondecreasing” (monotone) along every direction.

Example 2.8. Consider the diagonal matrix $\mathbf{M}$ defined by

$$\mathbf{M} = \begin{pmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_d \end{pmatrix},$$

where $\lambda_1, \dots, \lambda_d$ are nonnegative. Its eigenvalues are $\lambda_1, \dots, \lambda_d$, so it is PSD. Such matrices arise when the variables are “separated”, for instance as the Hessian of a function f of the form

$$f(\mathbf{x}) = \sum_{j=1}^d \frac{\lambda_j}{2} x_j^2,$$

which is therefore convex.

Symmetric matrices and quadratic forms The diagonal case of the previous example is, in fact, the general case, up to a rotation of the axes. This is the content of the spectral theorem, one of the most useful results of linear algebra (Strang, 2016; Horn and Johnson, 2012).

Proposition 2.8: Spectral theorem

Let $\mathbf{A} \in \mathbb{R}^{d \times d}$ be a symmetric matrix ($\mathbf{A}^T = \mathbf{A}$). Then all the eigenvalues $\lambda_1, \dots, \lambda_d$ of $\mathbf{A}$ are real, and there exists an orthonormal basis $(\mathbf{u}_1, \dots, \mathbf{u}_d)$ of $\mathbb{R}^d$ made of eigenvectors of $\mathbf{A}$ ($\mathbf{A}\mathbf{u}_j = \lambda_j\mathbf{u}_j$). In matrix form, with $\mathbf{U} = \begin{pmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_d \end{pmatrix}$ (so that $\mathbf{U}^T\mathbf{U} = \mathbf{I}$) and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_d)$,

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T = \sum_{j=1}^d \lambda_j \mathbf{u}_j \mathbf{u}_j^T.$$

Writing $\mathbf{z}$ in the eigenbasis, $\mathbf{z} = \sum_j c_j \mathbf{u}_j$ with $c_j = \langle \mathbf{z}, \mathbf{u}_j \rangle$, the quadratic form associated with $\mathbf{A}$ becomes a weighted sum of squares:

$$\mathbf{z}^T \mathbf{A} \mathbf{z} = \sum_{j=1}^d \lambda_j c_j^2.$$

This immediately shows that $\mathbf{A}$ is PSD ($\mathbf{z}^T \mathbf{A} \mathbf{z} \geq 0$ for all $\mathbf{z}$) if and only if all the λ_j are nonnegative, and positive definite if and only if they are all positive. It also gives a geometric picture, shown in Figure 17: the eigenvectors are the axes of the level sets of $\mathbf{z} \mapsto \mathbf{z}^T \mathbf{A} \mathbf{z}$, and the eigenvalues control how fast the function grows along each axis. When f is twice differentiable, Taylor’s formula $f(\mathbf{u} + \mathbf{z}) \approx f(\mathbf{u}) + \langle \nabla f(\mathbf{u}), \mathbf{z} \rangle + \frac{1}{2} \mathbf{z}^T \nabla^2 f(\mathbf{u}) \mathbf{z}$ shows that, locally, f looks like such a quadratic form: this is why the eigenvalues of the Hessian decide whether f is convex.

Example 2.9. Let $\mathbf{A} \in \mathbb{R}^{d \times d}$ be symmetric, $\mathbf{b} \in \mathbb{R}^d$, and $f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x}$. Then $\nabla f(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{b}$ and $\nabla^2 f(\mathbf{x}) = \mathbf{A}$ for every $\mathbf{x}$, so f is convex if and only if $\mathbf{A}$ is PSD.

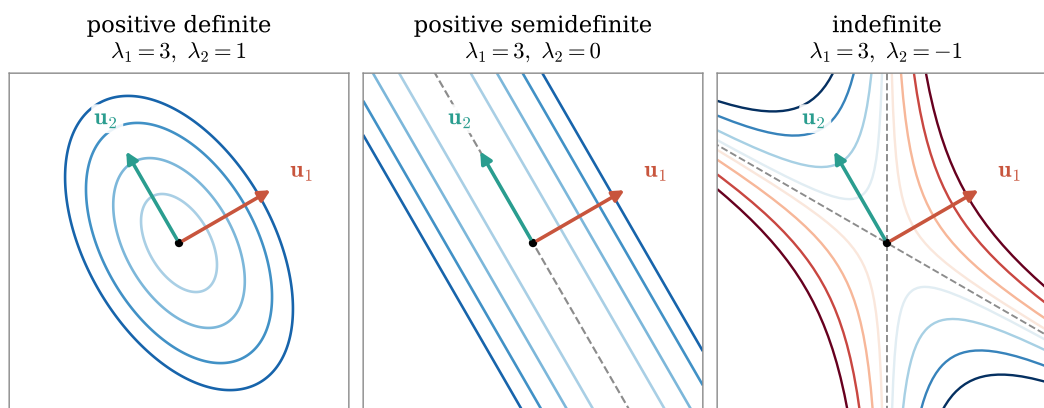


Figure 17: Level sets of the quadratic form $q(\mathbf{z}) = \mathbf{z}^T \mathbf{A} \mathbf{z}$ for $\mathbf{A} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^T$ with the same eigenvectors $\mathbf{u}_1, \mathbf{u}_2$ (arrows) and three choices of eigenvalues. Left: both eigenvalues are positive, the level sets are ellipses whose axes are the eigenvectors, longer along the direction of the smaller eigenvalue; q is strictly convex with a unique minimizer. Middle: one eigenvalue is zero, q is constant along $\mathbf{u}_2$ (a “valley” of minimizers); q is convex but not strictly. Right: the eigenvalues have opposite signs, q increases along $\mathbf{u}_1$ and decreases along $\mathbf{u}_2$ (red: positive values, blue: negative values); 0 is a saddle point and q is not convex. [code: make_figures_ch01.py: fig_ch01_quadratic_forms]

A fundamental instance is the least-squares objective $f(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$ of linear regression, whose Hessian $2\mathbf{X}^T \mathbf{X}$ is always PSD, since $\mathbf{z}^T \mathbf{X}^T \mathbf{X} \mathbf{z} = \|\mathbf{X}\mathbf{z}\|_2^2 \geq 0$: the least-squares problem is always convex.

Remark When the smallest eigenvalue of the Hessian is bounded away from zero, *i.e.* $\nabla^2 f(\mathbf{u}) \succeq \alpha \mathbf{I}$ for some $\alpha > 0$ and all $\mathbf{u}$, the function grows at least like a quadratic in every direction, as in the left panel of Figure 17. Such functions are called *strongly convex* (Definition 3.4 in the next section); they have a unique minimizer, and gradient methods converge faster on them. Adding a ridge penalty $\frac{\alpha}{2} \|\mathbf{w}\|_2^2$ to a convex objective makes it α -strongly convex, since it adds $\alpha \mathbf{I}$ to the Hessian.

To remember from Section 2

- A norm measures the size of a vector; the L_1 , L_2 and L_∞ norms have different unit balls, all convex. The inner product $\langle \mathbf{x}, \mathbf{y} \rangle = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta$ measures angles and gives orthogonal projections.
- The gradient $\nabla f(\mathbf{x}_0)$ gives the best linear approximation of f around $\mathbf{x}_0$ and points in the direction of steepest ascent; the Hessian $\nabla^2 f(\mathbf{x}_0)$ gives the second-order (quadratic) approximation.
- A set is convex if it contains every segment joining two of its points; a function is convex if its chords lie above its graph, *i.e.* if its epigraph is convex.
- For differentiable functions: f is convex $\iff$ its tangents lie below its graph $\iff$ its gradient is monotone $\iff$ (if twice differentiable) its Hessian is PSD everywhere. Consequently, every critical point of a differentiable

convex function is a global minimum.

- A symmetric matrix is diagonal in an orthonormal basis of eigenvectors; it is PSD if and only if its eigenvalues are nonnegative, and its eigenvectors and eigenvalues give the axes and the curvatures of the level sets of its quadratic form.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

Norms

- Q2.1.** Is the function $\mathbf{x} \mapsto \|\mathbf{x}\|_2^2$ a norm?
- Q2.2.** What is the orthogonal projection of $\mathbf{x} = (2, 3)^T$ onto the line spanned by $\mathbf{y} = (1, 0)^T$? And onto the line spanned by $(1, 1)^T$?
- Q2.3.** True or false? For every $\mathbf{x} \in \mathbb{R}^d$, $\|\mathbf{x}\|_1 \leq \|\mathbf{x}\|_2$.
- Q2.4.** Compute $\|\mathbf{I}_d\|_F$ and, for $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^d$, the Frobenius norm of the rank-one matrix $\mathbf{xy}^T$.

Derivatives

- Q2.5.** Compute the gradient and the Hessian of $f(\mathbf{x}) = \|\mathbf{x}\|_2^2$ on $\mathbb{R}^d$.
- Q2.6.** Let $f(x_1, x_2) = x_1^2 x_2$. Use the first-order approximation at $(1, 2)$ to estimate $f(1.01, 2)$, and compare with the exact value.
- Q2.7.** From the point $(1, 1)$, in which unit direction does $f(x_1, x_2) = x_1^2 + 3x_2^2$ decrease the fastest? Is it the direction pointing to the minimizer $(0, 0)$?

Convex sets

- Q2.8.** Is the union of two convex sets convex? And their intersection?
- Q2.9.** Is the set $\{\mathbf{x} \in \mathbb{R}^2 \mid x_1 x_2 \geq 1\}$ convex?
- Q2.10.** What is the convex hull of the five points $(0, 0), (1, 0), (0, 1), (1, 1)$ and $(\frac{1}{2}, \frac{1}{2})$? Is the way of writing $(\frac{1}{2}, \frac{1}{2})$ as a convex combination of the four others unique?

Synthesis of the section

- Q2.11.** Is $f(x) = |x|$ convex? Can Proposition 2.6 be used to prove it?
- Q2.12.** A differentiable convex function satisfies $\nabla f(\mathbf{u}^*) = \mathbf{0}$. What can you say about $\mathbf{u}^*$? Is it necessarily the unique minimizer?
- Q2.13.** Is the matrix $\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}$ positive definite?
- Q2.14.** Show that $f(x) = e^x - x$ is strictly convex on $\mathbb{R}$ and find its minimizer.
- Q2.15.** What happens if the Hessian of a twice differentiable function f has a negative eigenvalue at some point $\mathbf{u}$?
- Q2.16.** True or false? A convex function $f: \mathbb{R} \rightarrow \mathbb{R}$ that is bounded from above is constant.

3 Unconstrained and Smooth Convex Optimization

This section gives a general presentation of convex optimization; it then focuses on unconstrained convex optimization and presents some fundamental algorithms.

About this section

Guiding question. *How does a computer find the parameters that minimize a training loss when no closed-form formula is available?*

In this section you will learn to:

- characterize the critical points of a differentiable function with its gradient (Euler's equation) and classify them with its Hessian;
- write and run gradient descent, and understand the role of the step size, of line searches (exact step, Armijo's rule) and of the conditioning of the problem;
- use Newton's method and quasi-Newton methods (BFGS), and know when they are worth their cost;
- recognize the convergence speeds of these algorithms on a log-scale plot.

3.1 Motivation

Given a convex **function** $f : \mathbb{R}^d \rightarrow \mathbb{R}$, we want to solve the problem

$$\mathbf{x}^* = \arg \min_{\mathbf{x} \in \mathbb{R}^d} f(\mathbf{x}).$$

The aim of this section is to introduce algorithms that build a sequence $(\mathbf{x}_n)_{n \in \mathbb{N}}$ converging to $\mathbf{x}^*$, the point at which f reaches its minimum, as in Figure 18.

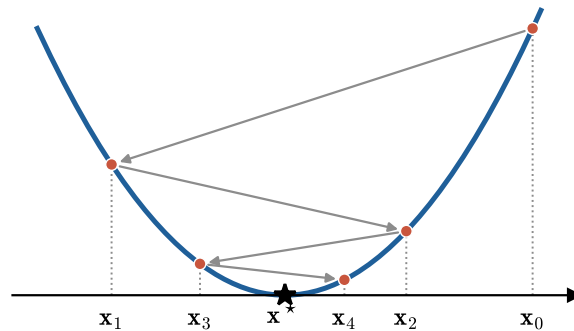


Figure 18: A descent algorithm builds a sequence of points $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$ along which the value of f decreases, and which converges to the minimizer $\mathbf{x}^*$ of f . [code: make_figures_ch02.py: fig_ch02_motivation]

We will see how to build such a sequence but, before doing so, we will see how to characterize the optimum of a function f (in our case, its minimum), *i.e.* what the optimality conditions are.

3.2 Optimality Conditions

Let us first define the minimum of a function f .

Definition 3.1: Minimum

Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a continuous function. We say that $\mathbf{u} \in \mathbb{R}^d$ is a **local minimum** of f if there exists a neighborhood $V \subset \mathbb{R}^d$ of $\mathbf{u}$ such that

$$f(\mathbf{u}) \leq f(\mathbf{v}), \quad \forall \mathbf{v} \in V.$$

We say that $\mathbf{u}$ is a **global minimum** of f if

$$f(\mathbf{u}) \leq f(\mathbf{v}), \quad \forall \mathbf{v} \in \mathbb{R}^d.$$

In other words, a minimum of the function f is simply a point $\mathbf{x}$ where f reaches its lowest value (locally or globally), as illustrated in Figure 19.

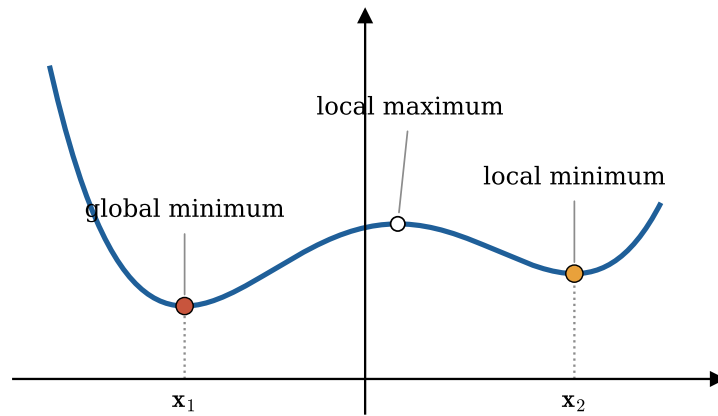


Figure 19: Local and global minima of a function of one variable. The point $\mathbf{x}_1$ is the global minimum of the function, while $\mathbf{x}_2$ is only a local minimum; between them lies a local maximum.

[code: make_figures_ch02.py: fig_ch02_minima]

Although such minima are easy to find graphically for a function of one variable, we need other criteria to detect them in general.

The following proposition gives a first characterization of relative (or local) minima.

Proposition 3.1: Euler's Inequality

Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a continuous function and $\mathcal{U}$ a nonempty convex set. Furthermore, let $\mathbf{u} \in \mathcal{U}$ be a relative minimum of f on $\mathcal{U}$. If f is differentiable at $\mathbf{u}$, then

$$\langle \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle \geq 0, \quad \forall \mathbf{v} \in \mathcal{U}.$$

This first result will be very useful for constrained problems (see Section 4), but

it is rarely used as such for unconstrained ones. If we go back to Figure 19, we see that the minima (and maxima) are located at the points where the derivative of the function is equal to 0. This optimality condition is known as Euler's equation.

Proposition 3.2: Euler's Equation

Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a continuous function which is differentiable at $\mathbf{u} \in \mathbb{R}^d$. If $\mathbf{u}$ is a local minimum of f , then

$$\nabla f(\mathbf{u}) = \mathbf{0}.$$

Proof. Let V be a neighborhood of $\mathbf{u}$ as in the definition of a local minimum. For all $\mathbf{v} \in \mathbb{R}^d$, $\mathbf{u} + t\mathbf{v} \in V$ as soon as $t > 0$ is small enough, and a first-order expansion gives

$$\begin{aligned} f(\mathbf{u}) &\leq f(\mathbf{u} + t\mathbf{v}) = f(\mathbf{u}) + t\langle \nabla f(\mathbf{u}), \mathbf{v} \rangle + t\|\mathbf{v}\| \varepsilon(t\mathbf{v}), \\ \Leftrightarrow 0 &\leq t\langle \nabla f(\mathbf{u}), \mathbf{v} \rangle + t\|\mathbf{v}\| \varepsilon(t\mathbf{v}), \end{aligned}$$

where $\varepsilon(t\mathbf{v}) \rightarrow 0$ when $t \rightarrow 0$. Dividing by $t > 0$ and letting $t \rightarrow 0$, we obtain

$$0 \leq \langle \nabla f(\mathbf{u}), \mathbf{v} \rangle.$$

Replacing $\mathbf{v}$ by $-\mathbf{v}$, we also have

$$0 \leq -\langle \nabla f(\mathbf{u}), \mathbf{v} \rangle.$$

Hence $\langle \nabla f(\mathbf{u}), \mathbf{v} \rangle = 0$ for all $\mathbf{v} \in \mathbb{R}^d$, and thus $\nabla f(\mathbf{u}) = \mathbf{0}$ (take $\mathbf{v} = \nabla f(\mathbf{u})$). $\square$

Although this proof relies on the definition of a (local or global) minimum, the same result holds for a maximum. The solutions of Euler's equation are called *critical points* (or *stationary points*).

Note that this proposition only provides a necessary condition for a point to be a local optimum; this condition is by no means sufficient, as the example below shows.

Example 3.1. Let us consider the functions f and g respectively defined by $f(x) = \frac{1}{2}(x-2)^2 - 2$ and $g(x) = \frac{1}{2}x^3 - 1$, and represented in Figure 20.

The points marked on both graphs ($x = 2$ and $x = 0$) are solutions of Euler's equation, $f'(x) = 0$ and $g'(x) = 0$ respectively. However, while this point is a minimum of f , this is no longer the case for g : its critical point is neither a minimum nor a maximum.

This example shows that it is important to characterize the different solutions of Euler's equation, *i.e.* to determine when they correspond to optima, and what kind of optima they are.

The solutions of *Euler's equation* contain all the points where a differentiable function f reaches a local extremum (a local or global minimum or maximum). Given

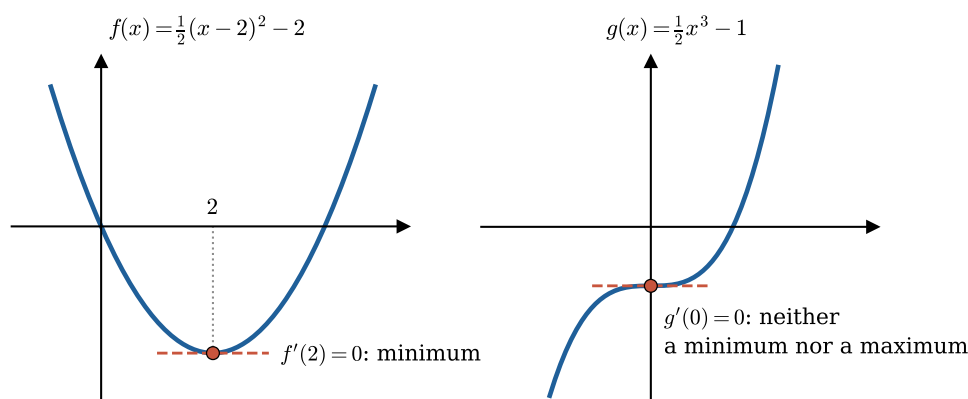


Figure 20: Euler's equation is only a necessary condition. Both functions have a horizontal tangent (dashed) at the marked point, but this critical point is a minimum of f (left) and neither a minimum nor a maximum of g (right). [code: make_figures_ch02.py: fig_ch02_eulerid]

a solution $\mathbf{u}$ of $\nabla f(\mathbf{u}) = \mathbf{0}$, and assuming that f is twice continuously differentiable, we can say that:

- $\mathbf{u}$ is a **local minimum** if $\nabla^2 f(\mathbf{u}) = \mathbf{H}_f(\mathbf{u}) \succ 0$, *i.e.* if the Hessian matrix evaluated at $\mathbf{u}$ is positive definite (all its eigenvalues are positive). This point is a global minimum if the function is **convex** everywhere; in fact, for a convex function, every critical point is a global minimum;
- $\mathbf{u}$ is a **local maximum** if $\nabla^2 f(\mathbf{u}) = \mathbf{H}_f(\mathbf{u}) \prec 0$, *i.e.* if the Hessian matrix evaluated at $\mathbf{u}$ is negative definite (all its eigenvalues are negative). This point is a global maximum if the function is **concave** everywhere;
- if $\nabla^2 f(\mathbf{u})$ has both positive and negative eigenvalues, $\mathbf{u}$ is neither a minimum nor a maximum (it is a *saddle point*);
- if $\nabla^2 f(\mathbf{u})$ is only semidefinite (some eigenvalues are zero), we cannot conclude without further study; think of $g(x) = \frac{1}{2}x^3 - 1$ above, for which $g''(0) = 0$.

These three typical situations are represented in Figure 21. They come from the second-order Taylor expansion around a critical point $\mathbf{u}$: since $\nabla f(\mathbf{u}) = \mathbf{0}$,

$$f(\mathbf{u} + \mathbf{h}) = f(\mathbf{u}) + \frac{1}{2}\mathbf{h}^T \nabla^2 f(\mathbf{u}) \mathbf{h} + o(\|\mathbf{h}\|^2),$$

so that, locally, f behaves like the quadratic form defined by its Hessian: it increases in the directions of the eigenvectors associated with positive eigenvalues and decreases in those associated with negative eigenvalues.

Pitfall In Machine Learning, a vanishing gradient does not mean that we have found a minimum. For nonconvex objectives, such as the training loss of a neural network, saddle points are much more frequent than local maxima in high dimension, and an algorithm based on the gradient alone may slow down considerably in their neighborhood. Convexity is precisely the property that rules out this difficulty: every critical point of a differentiable convex function is a global minimum.

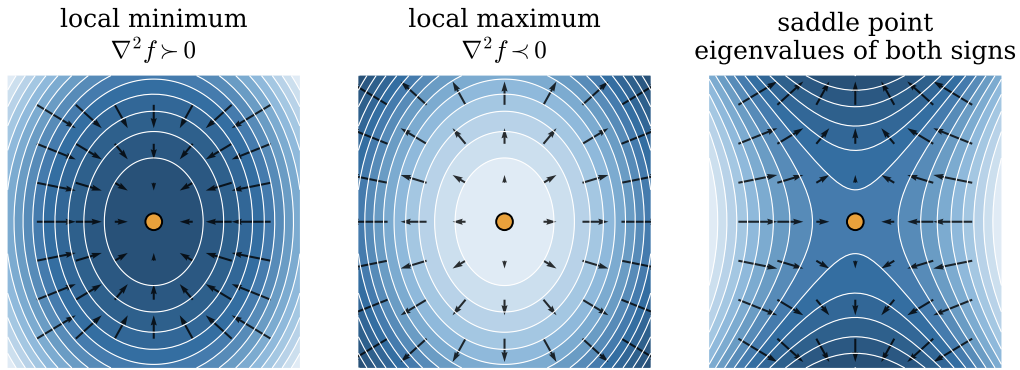


Figure 21: Level sets of f (shaded) and descent directions $-\nabla f$ (arrows) around the three types of nondegenerate critical points (orange dot), where $\nabla f = \mathbf{0}$. Gradient descent is attracted by a minimum, repelled by a maximum, and, around a saddle point, it is attracted along one direction and repelled along the other. [code: make_figures_ch02.py: fig_ch02_critical]

Example 3.2. Let us consider the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ defined by

$$f(\mathbf{x}) = x_1^2 + 4x_2^2 - 2x_1x_2.$$

We aim to find the extrema of this function.

We first compute the gradient ∇f of this function and find the values of $\mathbf{x}$ for which this gradient is equal to $\mathbf{0}$. The gradient is given by

$$\nabla f(\mathbf{x}) = \begin{pmatrix} 2x_1 - 2x_2 & 8x_2 - 2x_1 \end{pmatrix}.$$

Setting this gradient equal to $\mathbf{0}$ leads to $x_1 = x_2$ using the first component, and then to $x_1 = x_2 = 0$ using the second one. So the point $\mathbf{x}^* = (0, 0)$ is the only solution of Euler's equation.

We can now compute the Hessian matrix to determine what kind of extremum it is. The Hessian is given by

$$\nabla^2 f(\mathbf{x}) = \begin{pmatrix} 2 & -2 \\ -2 & 8 \end{pmatrix}.$$

The eigenvalues of this matrix, $5 \pm \sqrt{13}$, are both positive: the Hessian is positive definite at every point, so the function f is convex and the point $\mathbf{x}^*$ is its global minimum.

Example 3.3. Let us study the function $f : [-2, 2]^2 \rightarrow \mathbb{R}$ defined by

$$f(\mathbf{x}) = 4 + \left(x_1^2 - 2\cos(2\pi x_1)\right) + \left(x_2^2 - 2\cos(2\pi x_2)\right).$$

This function is particularly interesting because it has many local extrema but a single global minimum, as can be seen in Figure 22. It is therefore often used to test the efficiency of optimization algorithms.

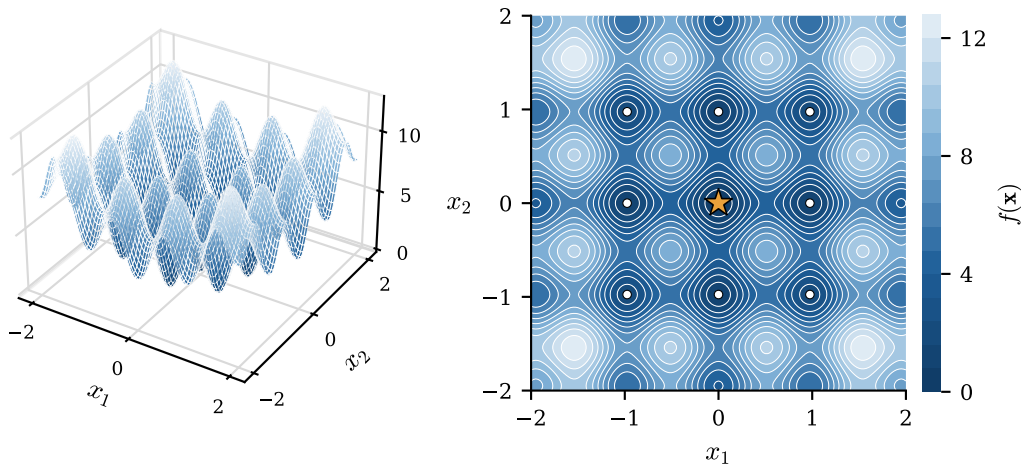


Figure 22: The function $f(\mathbf{x}) = 4 + (x_1^2 - 2 \cos(2\pi x_1)) + (x_2^2 - 2 \cos(2\pi x_2))$ on $[-2, 2]^2$. Left: its graph. Right: its level sets, some of its local minima (white dots) and its global minimum $\mathbf{0}$ (star). A gradient method started far from the origin is likely to stop at one of the local minima.
`[code: make_figures_ch02.py: fig_ch02_rastrigin]`

We can see that this function is neither convex nor concave, which could also be verified by studying its Hessian. The gradient of this function is easy to compute:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} 2x_1 + 4\pi \sin(2\pi x_1) & 2x_2 + 4\pi \sin(2\pi x_2) \end{pmatrix}.$$

The solutions of Euler's equation are obtained by solving the following system:

$$\begin{aligned} x_1 &= -2\pi \sin(2\pi x_1), \\ x_2 &= -2\pi \sin(2\pi x_2). \end{aligned}$$

The two equations are independent but remain hard to solve; an obvious solution, however, is given by the point $\mathbf{x}^* = (0, 0)$. We can also check that this point is the global minimum of our function: since $x_j^2 \geq 0$ and $-2 \cos(2\pi x_j) \geq -2$, we have $f(\mathbf{x}) \geq 0 = f(\mathbf{0})$, with equality only at $\mathbf{x} = \mathbf{0}$.

3.3 Minimization Problems

Given a vector space E of dimension d and a function $f : E \rightarrow \mathbb{R}$, an optimization problem consists in solving

$$\min_{\mathbf{x} \in E} f(\mathbf{x}).$$

The function f is sometimes called the **cost function**. It can represent, for instance, the cost for a company of storing a series of products (represented by the parameter $\mathbf{x}$), or the risk taken when making decisions.

Most of the time, we want to minimize this function f under some constraints; this will be studied in Section 4.

Let us first have a look at some basic optimization problems.

Linear Regression We are given a response vector $\mathbf{y} \in \mathbb{R}^m$ and a feature matrix $\mathbf{X} \in \mathbb{R}^{m \times d}$ whose rows are the feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_m \in \mathbb{R}^d$, where $d < m$. We would like to find a vector $\boldsymbol{\theta} \in \mathbb{R}^d$ that explains the values of $\mathbf{y}$ from $\mathbf{X}$ through the following model:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\varepsilon}, \quad \text{where } \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I}),$$

and where $\boldsymbol{\varepsilon}$ represents the errors of the model.

To find the best vector $\boldsymbol{\theta}$, we have to minimize these errors, *i.e.* to solve

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^d} \sum_{i=1}^m \varepsilon_i^2 = \min_{\boldsymbol{\theta} \in \mathbb{R}^d} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2.$$

We can easily check that this problem is convex. Indeed, the gradient of the objective function $F(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$ is

$$\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}),$$

and its second-order derivative is given by

$$\nabla_{\boldsymbol{\theta}}^2 F(\boldsymbol{\theta}) = 2\mathbf{X}^T \mathbf{X},$$

which is a symmetric positive semidefinite matrix of size d . Let us quickly show it: let λ be an eigenvalue of $\mathbf{X}^T \mathbf{X}$ and $\mathbf{u}_\lambda$ an associated eigenvector. By definition of an eigenvalue and an eigenvector, we have

$$\begin{aligned} \mathbf{X}^T \mathbf{X} \mathbf{u}_\lambda &= \lambda \mathbf{u}_\lambda \\ &\quad \downarrow \text{premultiplying by } \mathbf{u}_\lambda^T \\ \mathbf{u}_\lambda^T \mathbf{X}^T \mathbf{X} \mathbf{u}_\lambda &= \lambda \mathbf{u}_\lambda^T \mathbf{u}_\lambda \\ &\quad \downarrow \text{left: property of the transpose; right: definition of the norm} \\ (\mathbf{X} \mathbf{u}_\lambda)^T \mathbf{X} \mathbf{u}_\lambda &= \lambda \|\mathbf{u}_\lambda\|_2^2 \\ &\quad \downarrow \text{left: definition of the norm} \\ \|\mathbf{X} \mathbf{u}_\lambda\|_2^2 &= \lambda \|\mathbf{u}_\lambda\|_2^2. \end{aligned}$$

Since an eigenvector is necessarily nonzero, we have

$$\lambda = \frac{\|\mathbf{X} \mathbf{u}_\lambda\|_2^2}{\|\mathbf{u}_\lambda\|_2^2} \geq 0.$$

Finally, when $\mathbf{X}^T \mathbf{X}$ is invertible, the unique solution of Euler's equation $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) = \mathbf{0}$ is given by

$$\boldsymbol{\theta}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}.$$

In this case, an analytical solution of the minimization problem exists. Unfortunately, this is not always the case.

Logistic Regression Let us now consider the logistic regression problem, which is quite similar to the previous one. We want to find a model that predicts the class $y \in \{0, 1\}$ of our data (see Section 8.4 for further details).

To predict the class of an individual $\mathbf{x}$, we use a model of the form

$$g(\mathbf{x}, \boldsymbol{\theta}) = \log \left(\frac{\mathbb{P}(y = 1 \mid \mathbf{x})}{1 - \mathbb{P}(y = 1 \mid \mathbf{x})} \right) = \theta_0 + \theta_1 x_1 + \dots + \theta_d x_d.$$

The parameters of the model are estimated by minimizing the negative log-likelihood of our data:

$$\ell(\mathbf{X}, \boldsymbol{\theta}) = - \sum_{i=1}^m \left(y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right),$$

where

$$p_i = \frac{1}{1 + \exp \left(-\theta_0 - \sum_{j=1}^d \theta_j X_{ij} \right)}$$

is the probability, predicted by the model, that the i -th individual belongs to class 1.

Unfortunately, there is no analytical solution to this problem, so we need a way to approximate the solution step by step. This is the purpose of gradient descent algorithms.

3.4 Gradient Descent Algorithms

Let f be a function and $\mathcal{U}$ a nonempty set, and assume that the following problem has a solution:

$$\mathbf{u}^* = \arg \min_{\mathbf{v} \in \mathcal{U}} f(\mathbf{v}).$$

The idea is to build a sequence $(\mathbf{u}_k)_{k \in \mathbb{N}}$ which converges towards $\mathbf{u}^*$ using the following scheme:

- take an initial value $\mathbf{u}_0$: this is where the optimization procedure starts;
- $\mathbf{u}_k \rightarrow \mathbf{u}_{k+1}$: choose a direction $\mathbf{d}_k$ and minimize the function f along this direction;
- solve $\rho_k = \arg \min_{\rho > 0} f(\mathbf{u}_k - \rho \mathbf{d}_k)$: how far should we go in the selected direction?
- $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho_k \mathbf{d}_k$: update the current position.

This process is repeated until we reach the optimal solution $\mathbf{u}^*$.

Two questions then arise: (i) how should we choose the descent direction, and (ii) how far should we move in the chosen direction?

Indeed, as shown in Figure 23, several paths seem to lead to the desired solution, but is one of them preferable to the others? How can we use our knowledge of the function to be optimized to build such a path, rather than being guided by chance?

To answer the first question, we focus on the following quantity at a given iteration k :

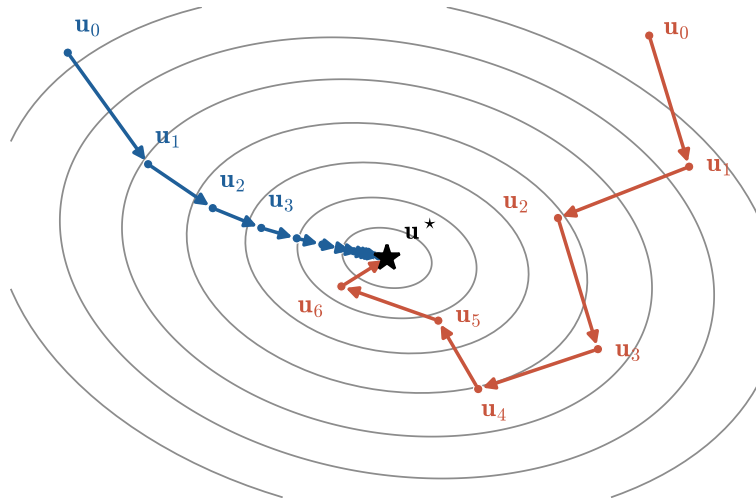


Figure 23: Two ways to reach the optimal solution $\mathbf{u}^*$ of the function f we aim to minimize; the ellipses represent the level sets of f . The red path uses arbitrary directions; the blue path follows, at each iteration, the direction of steepest descent $-\nabla f(\mathbf{u}_k)$, which is orthogonal to the level set going through $\mathbf{u}_k$ (gradient descent with constant step). [code: make_figures_ch02.py: fig_ch02_paths]

$$f(\mathbf{u}_k - \rho \mathbf{d}_k).$$

More precisely, we focus on the first-order approximation of this quantity. Remember that we have the following expansion when ρ is close to 0:

$$f(\mathbf{u}_k - \rho \mathbf{d}_k) = f(\mathbf{u}_k) - \rho \langle \nabla f(\mathbf{u}_k), \mathbf{d}_k \rangle + \rho \varepsilon(\rho),$$

where ε is a function that tends to 0 when ρ tends to 0. To decrease $f(\mathbf{u}_k - \rho \mathbf{d}_k)$ as much as possible, we have to choose, among the directions $\mathbf{d}_k$ of a given norm, the one that maximizes the inner product $\langle \nabla f(\mathbf{u}_k), \mathbf{d}_k \rangle$. By the **Cauchy-Schwarz inequality**, this direction is given by $\mathbf{d}_k = \nabla f(\mathbf{u}_k)$ (up to a positive factor): the opposite of the gradient is the direction of steepest descent.

In a nutshell Imagine a hiker lost in the fog on a mountain, who wants to reach the bottom of the valley. She cannot see the landscape, but she can feel the slope under her feet: she takes a step in the direction where the ground goes down most steeply, then looks again. The gradient is the local slope, and gradient descent only uses this local information; the step size says how long each stride is.

So the previous algorithm becomes:

- choose $\mathbf{u}_0$ to initialize the algorithm;
- set $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho_k \nabla f(\mathbf{u}_k)$ with $\rho_k > 0$.

However, we also need a stopping criterion. To find one, keep in mind that the critical points $\mathbf{u}^*$ satisfy $\nabla f(\mathbf{u}^*) = \mathbf{0}$. Thus, when the norm of the gradient of f is close

to 0, we can assume that we have reached an (or the) optimal solution. In practice, we choose a small enough constant $\eta > 0$ and apply the previous algorithm until

$$\|\nabla f(\mathbf{u}_k)\| \leq \eta.$$

It remains to choose the value of ρ_k at each step of the gradient descent algorithm. The simplest choice is a constant step, *i.e.* $\rho_k = \rho > 0$ for all k .

This gives our first descent algorithm.

Definition 3.2: Gradient Descent with Constant Step

Let f be a differentiable function defined on a vector space E of dimension d , and let ρ, η be two positive real numbers. Gradient Descent with Constant Step is defined as follows:

- choose $\mathbf{u}_0$ to initialize the algorithm, and set $k = 0$;
- while $\|\nabla f(\mathbf{u}_k)\| > \eta$:
 1. compute $\nabla f(\mathbf{u}_k)$,
 2. set $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho \nabla f(\mathbf{u}_k)$,
 3. set $k \leftarrow k + 1$;
- return $\mathbf{u}_k$.

However, a constant step is not always the best choice in practice. Indeed, a bad choice of this constant can lead to a very slow convergence of the algorithm, or even to its divergence, as illustrated in Figure 24. For instance, if f is α -strongly convex (see Definition 3.4 below) and if its gradient is L -Lipschitz, gradient descent with constant step converges as soon as $0 < \rho < 2\alpha/L^2$.

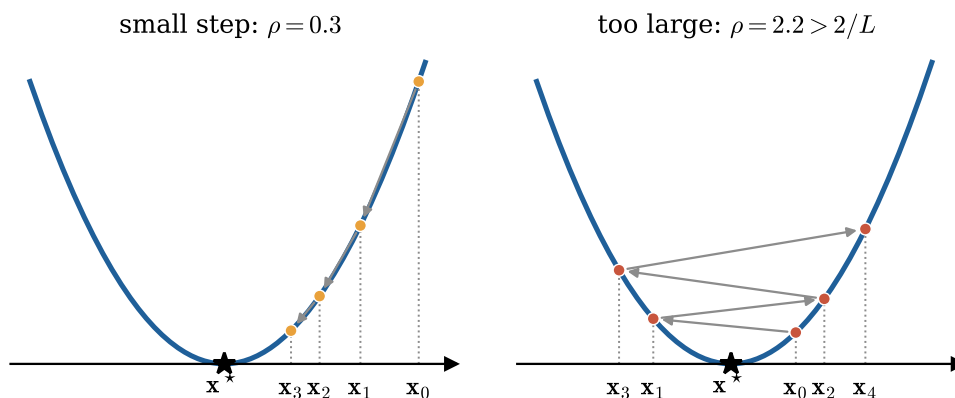


Figure 24: Importance of the choice of the learning rate ρ for gradient descent on $f(x) = \frac{1}{2}x^2$, for which $L = 1$ and $x_{k+1} = (1 - \rho)x_k$. A small value slows down the convergence of the algorithm (left), while a too large value, $\rho > 2/L$, makes the algorithm diverge (right). [code: make_figures_ch02.py: fig_ch02_steps1d]

The same phenomenon in dimension 2 is shown in Figure 25, on the quadratic

function $f(\mathbf{u}) = \frac{1}{2}(u_1^2 + 10u_2^2)$. The curvature is ten times larger along u_2 than along u_1 : the step must be small enough for the steep direction, which makes the progress slow along the flat one.

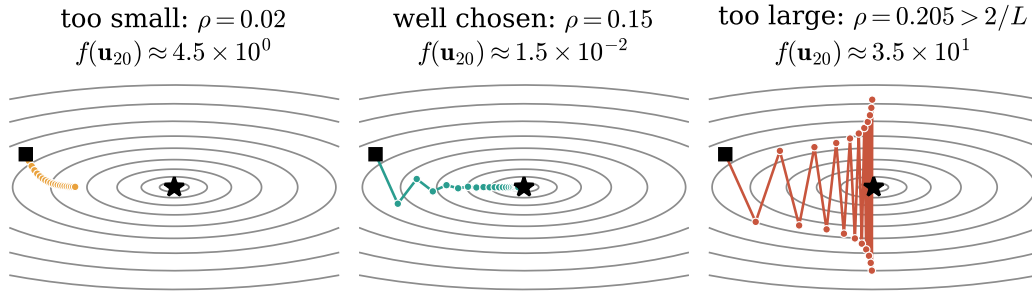


Figure 25: Twenty iterations of gradient descent with constant step on $f(\mathbf{u}) = \frac{1}{2}(u_1^2 + 10u_2^2)$, for which $\alpha = 1$ and $L = 10$, from the same starting point (black square). Left: the step is too small and the iterates crawl towards $\mathbf{u}^*$. Middle: a good step. Right: $\rho > 2/L$, the iterates oscillate with increasing amplitude along the steep direction and the algorithm diverges. [code: `make_figures_ch02.py: fig_ch02_stepsizes`]

Remark on the choice of a constant step A function whose gradient is L -Lipschitz, $\|\nabla f(\mathbf{u}) - \nabla f(\mathbf{v})\| \leq L \|\mathbf{u} - \mathbf{v}\|$, is said to be **L -smooth**; for a twice differentiable function, this means that the eigenvalues of the Hessian are bounded by L in absolute value. The constant L is the largest curvature of f , and it gives the natural scale of the step: the classical choice is $\rho = 1/L$. The condition $\rho < 2\alpha/L^2$ given above is sufficient but pessimistic. For a convex and L -smooth function, gradient descent converges for every $\rho \in]0, 2/L[$ (Nesterov, 2018, Section 2.1.5); with $\rho = 1/L$,

$$f(\mathbf{u}_k) - f(\mathbf{u}^*) \leq \frac{L \|\mathbf{u}_0 - \mathbf{u}^*\|^2}{2k},$$

and, if f is moreover α -strongly convex, $\|\mathbf{u}_k - \mathbf{u}^*\|^2 \leq (1 - \alpha/L)^k \|\mathbf{u}_0 - \mathbf{u}^*\|^2$: the error decreases geometrically, a behavior called *linear convergence* because it appears as a straight line on a log-scale plot. On a quadratic function, the threshold $2/L$ is sharp, as illustrated in Figure 25.

We may rather choose a nonconstant step ρ_k that decreases with k , *e.g.* $\rho_k = \frac{1}{k}$. The most natural choice, however, is the value ρ_k which minimizes the function along the given direction.

This leads to another gradient descent algorithm, with a so-called *optimal step*.

Definition 3.3: Gradient Descent with Optimal Step

Let f be a differentiable function defined on a vector space E of dimension d , and let η be a positive real number. Gradient Descent with Optimal Step is defined as follows:

- choose $\mathbf{u}_0$ to initialize the algorithm, and set $k = 0$;

- while $\|\nabla f(\mathbf{u}_k)\| > \eta$:
 1. compute $\nabla f(\mathbf{u}_k)$,
 2. compute $\rho_k = \arg \min_{\rho > 0} f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))$,
 3. set $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho_k \nabla f(\mathbf{u}_k)$,
 4. set $k \leftarrow k + 1$;
- return $\mathbf{u}_k$.

One can show that this algorithm converges for *strongly convex functions*. We have already discussed convexity but not strong convexity, so let us define it.

Definition 3.4: Strong Convexity

Let f be a convex and continuously differentiable function on $\mathbb{R}^d$. We say that f is **strongly convex** (or **α -elliptic**) if there exists $\alpha > 0$ such that

$$\langle \nabla f(\mathbf{v}) - \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle \geq \alpha \|\mathbf{v} - \mathbf{u}\|^2, \quad \forall \mathbf{u}, \mathbf{v} \in \mathbb{R}^d.$$

Equivalently, when f is twice differentiable, f is α -strongly convex if and only if

$$\nabla^2 f(\mathbf{x}) \succeq \alpha \mathbf{I} \quad \text{for all } \mathbf{x} \in \mathbb{R}^d,$$

$$i.e. \text{ for all } \mathbf{x}, \mathbf{z} \in \mathbb{R}^d, \quad \mathbf{z}^T \nabla^2 f(\mathbf{x}) \mathbf{z} \geq \alpha \|\mathbf{z}\|^2.$$

Proposition 3.3: Optimal Step: Convergence

If f is α -strongly convex and if its gradient is Lipschitz continuous on every bounded set (for instance, if f is twice continuously differentiable), then Gradient Descent with Optimal Step converges to the unique minimizer $\mathbf{u}^*$ of f , whatever the initial point $\mathbf{u}_0$.

Let us now have a look at the gradient at each step. What can we say about $\langle \nabla f(\mathbf{u}_{k+1}), \nabla f(\mathbf{u}_k) \rangle$ when $\rho_k = \arg \min_{\rho > 0} f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))$?

If ρ_k minimizes $\rho \mapsto f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))$, then

$$\begin{aligned} & \frac{\partial}{\partial \rho} f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))|_{\rho=\rho_k} = 0 \\ \iff & -\langle \nabla f(\mathbf{u}_k - \rho_k \nabla f(\mathbf{u}_k)), \nabla f(\mathbf{u}_k) \rangle = 0 \\ \iff & \langle \nabla f(\mathbf{u}_{k+1}), \nabla f(\mathbf{u}_k) \rangle = 0. \end{aligned}$$

We can see that two successive descent directions are orthogonal. This observation is very useful to determine the optimal step for an important class of problems in Machine Learning, namely quadratic problems, as illustrated in the following example.

Example 3.4. Let $\mathbf{A}$ be a symmetric positive definite matrix and $\mathbf{b} \in \mathbb{R}^d$. Let us consider the quadratic form f defined for all $\mathbf{u} \in \mathbb{R}^d$ by

$$f(\mathbf{u}) = \frac{1}{2} \langle \mathbf{A}\mathbf{u}, \mathbf{u} \rangle - \langle \mathbf{b}, \mathbf{u} \rangle.$$

We aim to determine the optimal step at each iteration of gradient descent applied to this function. We first compute the gradient: for all $k \in \mathbb{N}$, we have $\nabla f(\mathbf{u}_k) = \mathbf{A}\mathbf{u}_k - \mathbf{b}$. Furthermore,

$$\begin{aligned} \nabla f(\mathbf{u}_{k+1}) &= \mathbf{A}\mathbf{u}_{k+1} - \mathbf{b} \\ &= \mathbf{A}(\mathbf{u}_k - \rho_k(\mathbf{A}\mathbf{u}_k - \mathbf{b})) - \mathbf{b} \\ &= \mathbf{A}\mathbf{u}_k - \mathbf{b} - \rho_k \mathbf{A}(\mathbf{A}\mathbf{u}_k - \mathbf{b}). \end{aligned}$$

Now, we use the fact that $\langle \nabla f(\mathbf{u}_{k+1}), \nabla f(\mathbf{u}_k) \rangle = 0$ for this algorithm:

$$\begin{aligned} \implies \langle \mathbf{A}\mathbf{u}_k - \mathbf{b}, \mathbf{A}\mathbf{u}_k - \mathbf{b} - \rho_k \mathbf{A}(\mathbf{A}\mathbf{u}_k - \mathbf{b}) \rangle &= 0 \\ \implies \langle \mathbf{A}\mathbf{u}_k - \mathbf{b}, \mathbf{A}\mathbf{u}_k - \mathbf{b} \rangle &= \rho_k \langle \mathbf{A}\mathbf{u}_k - \mathbf{b}, \mathbf{A}(\mathbf{A}\mathbf{u}_k - \mathbf{b}) \rangle \\ \implies \rho_k &= \frac{\langle \mathbf{A}\mathbf{u}_k - \mathbf{b}, \mathbf{A}\mathbf{u}_k - \mathbf{b} \rangle}{\langle \mathbf{A}\mathbf{u}_k - \mathbf{b}, \mathbf{A}(\mathbf{A}\mathbf{u}_k - \mathbf{b}) \rangle}. \end{aligned}$$

Finally, for quadratic problems of this form, the optimal learning rate at step k is equal to

$$\rho_k = \frac{\|\mathbf{A}\mathbf{u}_k - \mathbf{b}\|^2}{\|\mathbf{A}\mathbf{u}_k - \mathbf{b}\|_{\mathbf{A}}^2},$$

where $\|\mathbf{u}\|_{\mathbf{A}}^2 = \mathbf{u}^T \mathbf{A} \mathbf{u}$ (the denominator is positive as long as $\nabla f(\mathbf{u}_k) \neq \mathbf{0}$, since $\mathbf{A}$ is positive definite).

Because successive directions are orthogonal, gradient descent with optimal step follows a characteristic *zig-zag* path, shown in Figure 26. Its speed depends on the shape of the level sets: on the quadratic function of the previous example, one can show that

$$f(\mathbf{u}_{k+1}) - f(\mathbf{u}^*) \leq \left(\frac{\kappa - 1}{\kappa + 1} \right)^2 (f(\mathbf{u}_k) - f(\mathbf{u}^*)), \quad \text{where } \kappa = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$$

is the **condition number** of $\mathbf{A}$ (Nocedal and Wright, 2006, Section 3.3). When $\kappa = 1$ (circular level sets), the gradient points towards $\mathbf{u}^*$ and one step is enough; when κ is large (very elongated level sets), the factor is close to 1 and the convergence is slow.

In a nutshell The condition number $\kappa = L/\alpha$ (ratio of the largest to the smallest curvature) measures how elongated the valley of f is. Gradient descent needs a number of iterations roughly proportional to κ to reach a given accuracy. Many practical tricks in Machine Learning, such as standardizing the features or adding a ridge penalty $\lambda \|\boldsymbol{\theta}\|^2$ (which increases α by 2λ), can be seen as ways to reduce κ .

Other algorithms, or other ways to choose the learning rate at each step, can be studied, such as:

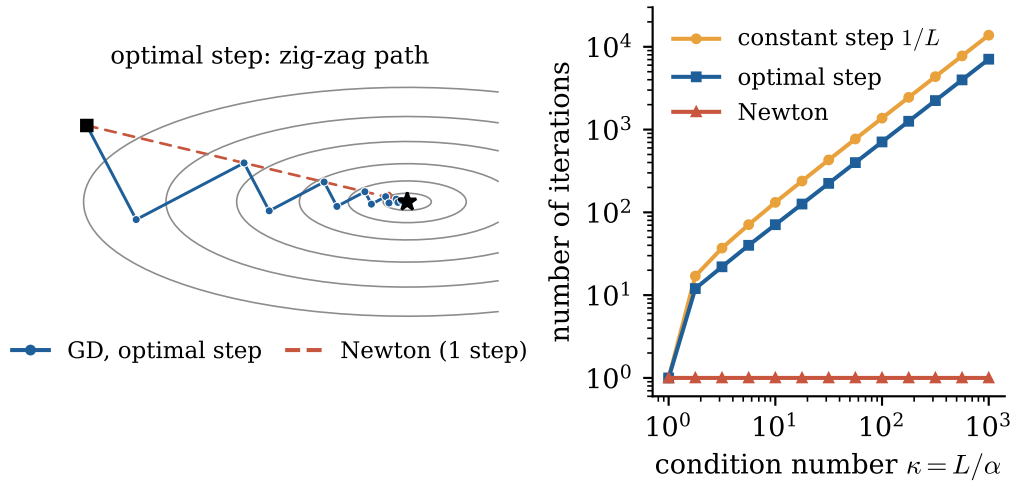


Figure 26: Left: gradient descent with optimal step on $f(\mathbf{u}) = \frac{1}{2}(u_1^2 + 8u_2^2)$; two successive directions are orthogonal, which produces a zig-zag path, whereas Newton's method (dashed) reaches the minimizer of a quadratic function in a single step. Right: number of iterations needed to reach $\|\nabla f(\mathbf{u}_k)\| \leq 10^{-6}$ on $f(\mathbf{u}) = \frac{1}{2}(u_1^2 + \kappa u_2^2)$, as a function of the condition number κ : it grows linearly with κ for gradient descent, and does not depend on κ for Newton's method.
[`code: make_figures_ch02.py: fig_ch02_zigzag`]

- **line search:** Armijo's criterion, Wolfe's criteria;
- **algorithms:** the conjugate gradient method.

Backtracking line search (Armijo's rule) Except for quadratic functions, the optimal step is itself the solution of an optimization problem, and computing it exactly at each iteration is too costly. In practice, we are satisfied with a step that decreases f sufficiently. Let $\varphi(\rho) = f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))$; note that $\varphi'(0) = -\|\nabla f(\mathbf{u}_k)\|^2 < 0$. Given $c \in]0, 1[$ (typically $c = 10^{-4}$), the step ρ satisfies **Armijo's condition** (Armijo, 1966) if

$$f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k)) \leq f(\mathbf{u}_k) - c \rho \|\nabla f(\mathbf{u}_k)\|^2,$$

i.e. if the graph of φ lies below the line of slope $c \varphi'(0)$ going through $(0, \varphi(0))$, see Figure 27. *Backtracking* starts from a large step $\rho = \rho_0$ and divides it by 2 (or multiplies it by some $\beta \in]0, 1[$) until Armijo's condition holds. Since $c < 1$, this condition always holds for ρ small enough, so backtracking stops after a finite number of trials; it requires only evaluations of f , and it frees us from the knowledge of L .

We conclude this part with a last algorithm called *Newton's method*.

Newton's method is also a descent algorithm. It uses the second-order derivative to refine the descent direction as follows:

$$\mathbf{u}_{k+1} \leftarrow \mathbf{u}_k - \left(\nabla^2 f(\mathbf{u}_k) \right)^{-1} \nabla f(\mathbf{u}_k).$$

- It requires fewer iterations to converge to the solution than the other gradient descent methods.
- It is harder to implement and requires inverting the Hessian of the function we want to optimize (in $\mathcal{O}(d^3)$ operations).

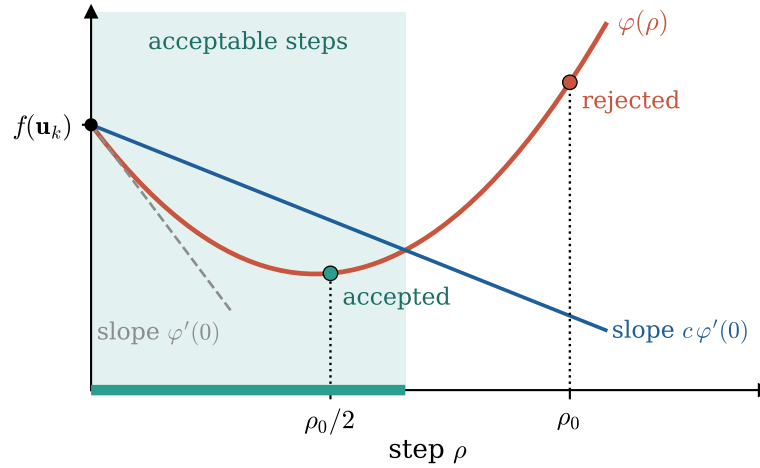


Figure 27: Armijo's rule. A step ρ is accepted (shaded region) when $\varphi(\rho)$ lies below the blue line, whose slope is a fraction c of the initial slope $\varphi'(0)$ (dashed), which guarantees a sufficient decrease of f . Backtracking tries ρ_0 , which is too large, then $\rho_0/2$, which is accepted. (Here $\varphi(\rho) = f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))$ is a parabola and $c = 0.3$ for readability; much smaller values of c are used in practice.) [code: make_figures_ch02.py: fig_ch02_armijo]

- The Hessian is not always invertible at a given point.

Newton's method is obtained by minimizing, at each iteration, the second-order approximation of f around $\mathbf{u}_k$:

$$f(\mathbf{u}_k + \mathbf{h}) \approx f(\mathbf{u}_k) + \nabla f(\mathbf{u}_k)^T \mathbf{h} + \frac{1}{2} \mathbf{h}^T \nabla^2 f(\mathbf{u}_k) \mathbf{h},$$

whose gradient with respect to $\mathbf{h}$ vanishes at $\mathbf{h} = -(\nabla^2 f(\mathbf{u}_k))^{-1} \nabla f(\mathbf{u}_k)$. For a quadratic function, this approximation is exact, and Newton's method finds the minimizer in a single iteration, whatever the condition number (Figure 26). For a general function, its convergence is *quadratic* near a minimizer at which the Hessian is positive definite: roughly speaking, the number of correct digits doubles at each iteration (Nocedal and Wright, 2006, Section 3.3). In practice, the system $\nabla^2 f(\mathbf{u}_k) \mathbf{h} = -\nabla f(\mathbf{u}_k)$ is solved (*e.g.* by a Cholesky factorization) rather than computing the inverse of the Hessian.

Pitfall Newton's method is only *locally* convergent. Far from the solution, the pure iteration (with step 1) can diverge, even for a convex function: for $f(x) = \sqrt{1 + x^2}$, we have $f'(x)/f''(x) = x(1 + x^2)$, so that $x_{k+1} = -x_k^3$, which diverges as soon as $|x_0| > 1$. Moreover, when $\nabla^2 f(\mathbf{u}_k)$ is not positive definite (nonconvex function), the Newton direction may not even be a descent direction, and the method can be attracted by a maximum or a saddle point. This is why Newton's method is used in practice with a line search (*damped Newton*), and why its first iterations are sometimes replaced by gradient steps.

The main drawback of Newton's method is the computation of the inverse of the Hessian matrix $\mathbf{H}_k^{-1}$, where $\mathbf{H}_k = \nabla^2 f(\mathbf{u}_k)$. To avoid it, quasi-Newton methods use the following iterations:

$$\begin{aligned}\mathbf{u}_{k+1} &= \mathbf{u}_k - \rho_k \mathbf{M}_k \nabla f(\mathbf{u}_k), \\ \mathbf{M}_{k+1} &= \mathbf{M}_k + \mathbf{C}_k.\end{aligned}$$

The idea is to approximate $\mathbf{H}_k^{-1}$ by a matrix $\mathbf{M}_k$, to which a correction matrix $\mathbf{C}_k$ is added at each step.

Remember that

$$\nabla f(\mathbf{u}_k) = \nabla f(\mathbf{u}_{k+1} + (\mathbf{u}_k - \mathbf{u}_{k+1})) \approx \nabla f(\mathbf{u}_{k+1}) + \nabla^2 f(\mathbf{u}_{k+1})(\mathbf{u}_k - \mathbf{u}_{k+1});$$

we then have the following approximation:

$$\left(\nabla^2 f(\mathbf{u}_{k+1})\right)^{-1} (\nabla f(\mathbf{u}_{k+1}) - \nabla f(\mathbf{u}_k)) \approx \mathbf{u}_{k+1} - \mathbf{u}_k.$$

If we set $\boldsymbol{\gamma}_k = \nabla f(\mathbf{u}_{k+1}) - \nabla f(\mathbf{u}_k)$ and $\boldsymbol{\delta}_k = \mathbf{u}_{k+1} - \mathbf{u}_k$, and if we require the new matrix $\mathbf{M}_{k+1}$, which plays the role of $(\nabla^2 f(\mathbf{u}_{k+1}))^{-1}$, to satisfy this relation exactly, we get the **quasi-Newton condition**:

$$\boxed{\mathbf{M}_{k+1} \boldsymbol{\gamma}_k = \boldsymbol{\delta}_k}.$$

It remains to see how to define such a correction matrix $\mathbf{C}_k$. We present two solutions, based on rank-1 and rank-2 matrices respectively, together with the corresponding algorithms. Note that, in practice, the rank-2 correction is used, in the optimization algorithm known as BFGS.

Rank-1 Correction The correction matrix $\mathbf{C}_k$ is assumed to be of rank 1 (and symmetric). So we can write $\mathbf{C}_k$ as $\mathbf{v}_k \mathbf{v}_k^T$, where $\mathbf{v}_k \in \mathbb{R}^d$.

The update rule of the matrix $\mathbf{M}_k$ can be written as

$$\mathbf{M}_{k+1} = \mathbf{M}_k + \mathbf{v}_k \mathbf{v}_k^T,$$

and the quasi-Newton condition gives

$$\begin{aligned}(\mathbf{M}_k + \mathbf{v}_k \mathbf{v}_k^T) \boldsymbol{\gamma}_k &= \boldsymbol{\delta}_k, \\ \mathbf{M}_k \boldsymbol{\gamma}_k + \mathbf{v}_k \mathbf{v}_k^T \boldsymbol{\gamma}_k &= \boldsymbol{\delta}_k, \\ \mathbf{v}_k (\mathbf{v}_k^T \boldsymbol{\gamma}_k) &= \boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k, \\ \mathbf{v}_k &= \frac{\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k}{\mathbf{v}_k^T \boldsymbol{\gamma}_k}.\end{aligned}$$

Moreover, taking the inner product of the third line with $\boldsymbol{\gamma}_k$ gives $(\mathbf{v}_k^T \boldsymbol{\gamma}_k)^2 = \boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k - \boldsymbol{\gamma}_k^T \mathbf{M}_k \boldsymbol{\gamma}_k = (\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)^T \boldsymbol{\gamma}_k$, hence

$$\mathbf{v}_k \mathbf{v}_k^T = \frac{(\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)(\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)^T}{(\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)^T \boldsymbol{\gamma}_k}.$$

This rank-one correction leads to the *Broyden algorithm* (also known as the symmetric rank-one, or SR1, method).

Definition 3.5: Broyden Algorithm

Let f be a differentiable function defined on a vector space E of dimension d , and let η be a positive real number. The Broyden algorithm is defined as follows:

- choose $\mathbf{u}_0$ to initialize the algorithm, and set $\mathbf{M}_0 = \mathbf{I}$ and $k = 0$;
- while $\|\nabla f(\mathbf{u}_k)\| > \eta$:
 1. set $\rho_k = \arg \min_{\rho \in \mathbb{R}} f(\mathbf{u}_k - \rho \mathbf{M}_k \nabla f(\mathbf{u}_k))$,
 2. set $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho_k \mathbf{M}_k \nabla f(\mathbf{u}_k)$,
 3. set $\mathbf{M}_{k+1} = \mathbf{M}_k + \frac{(\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)(\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)^T}{(\boldsymbol{\delta}_k - \mathbf{M}_k \boldsymbol{\gamma}_k)^T \boldsymbol{\gamma}_k}$,
 4. set $k \leftarrow k + 1$;
- return $\mathbf{u}_k$.

Rank-2 Correction The idea is quite similar but, instead of a rank-one matrix, the correction is written as the sum of two rank-one matrices, which leads to a rank-2 correction. It is easier to first derive the update of the approximation $\mathbf{B}_k$ of the Hessian $\nabla^2 f(\mathbf{u}_k)$ itself (so that $\mathbf{M}_k = \mathbf{B}_k^{-1}$), which must satisfy the equivalent quasi-Newton condition $\mathbf{B}_{k+1} \boldsymbol{\delta}_k = \boldsymbol{\gamma}_k$. We look for a correction of the form

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \alpha \mathbf{v}_k \mathbf{v}_k^T + \beta \mathbf{w}_k \mathbf{w}_k^T.$$

In order to determine suitable values of α and β , we set $\mathbf{v}_k = \boldsymbol{\gamma}_k$ and $\mathbf{w}_k = \mathbf{B}_k \boldsymbol{\delta}_k$ and use the quasi-Newton condition:

$$\mathbf{B}_k \boldsymbol{\delta}_k + \alpha (\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k) \boldsymbol{\gamma}_k + \beta (\boldsymbol{\delta}_k^T \mathbf{B}_k \boldsymbol{\delta}_k) \mathbf{B}_k \boldsymbol{\delta}_k = \boldsymbol{\gamma}_k.$$

This condition holds if

$$\alpha = \frac{1}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \quad \text{and} \quad \beta = -\frac{1}{\boldsymbol{\delta}_k^T \mathbf{B}_k \boldsymbol{\delta}_k},$$

and the update rule becomes

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\boldsymbol{\gamma}_k \boldsymbol{\gamma}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} - \frac{\mathbf{B}_k \boldsymbol{\delta}_k \boldsymbol{\delta}_k^T \mathbf{B}_k^T}{\boldsymbol{\delta}_k^T \mathbf{B}_k \boldsymbol{\delta}_k}.$$

Inverting this relation with the Sherman–Morrison–Woodbury formula gives the corresponding update of $\mathbf{M}_k = \mathbf{B}_k^{-1}$, which directly approximates the inverse of the Hessian and still satisfies $\mathbf{M}_{k+1} \boldsymbol{\gamma}_k = \boldsymbol{\delta}_k$:

$$\mathbf{M}_{k+1} = \left(\mathbf{I} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \right) \mathbf{M}_k \left(\mathbf{I} - \frac{\boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \right) + \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}.$$

The use of this update rule is known as the *Broyden–Fletcher–Goldfarb–Shanno* (BFGS) algorithm.

Definition 3.6: BFGS Algorithm

Let f be a differentiable function defined on a vector space E of dimension d , and let η be a positive real number. The BFGS algorithm is defined as follows:

- choose $\mathbf{u}_0$ to initialize the algorithm, and set $\mathbf{M}_0 = \mathbf{I}$ and $k = 0$;
- while $\|\nabla f(\mathbf{u}_k)\| > \eta$:
 1. set $\rho_k = \arg \min_{\rho \in \mathbb{R}} f(\mathbf{u}_k - \rho \mathbf{M}_k \nabla f(\mathbf{u}_k))$,
 2. set $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho_k \mathbf{M}_k \nabla f(\mathbf{u}_k)$,
 3. set $\mathbf{M}_{k+1} = \left(\mathbf{I} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \right) \mathbf{M}_k \left(\mathbf{I} - \frac{\boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \right) + \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}$,
 4. set $k \leftarrow k + 1$;
- return $\mathbf{u}_k$.

In practice, the exact minimization along the direction is replaced by a line search such as Armijo’s rule (usually combined with a curvature condition, the Wolfe conditions, which guarantees $\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k > 0$ and thus that $\mathbf{M}_{k+1}$ remains positive definite). BFGS combines the advantages of the two previous families: each iteration only costs $\mathcal{O}(d^2)$ operations and requires only gradients, as for gradient descent, while its convergence is *superlinear* near the solution, almost as fast as Newton’s method. For large d , storing the $d \times d$ matrix $\mathbf{M}_k$ itself becomes too expensive, and the limited-memory variant *L-BFGS*, which only keeps the last few pairs $(\boldsymbol{\delta}_k, \boldsymbol{\gamma}_k)$, is used instead (Nocedal and Wright, 2006, Chapter 7); it is, for instance, the default solver of logistic regression in scikit-learn.

Figure 28 compares the algorithms of this section on a logistic regression problem. On a log-scale plot, the error of gradient descent decreases along a straight line (linear convergence), that of BFGS along a curve that bends down (superlinear convergence), and that of Newton’s method falls vertically after a few iterations (quadratic convergence).

Pitfall The behavior of gradient descent depends on the units in which the features are measured (right panel of Figure 28), and so does the stopping criterion $\|\nabla f(\mathbf{u}_k)\| \leq \eta$: multiplying f by 100 multiplies its gradient by 100 without changing its minimizer. Always standardize the features before running a first-order method, and choose the tolerance η relative to the initial gradient, *e.g.* $\|\nabla f(\mathbf{u}_k)\| \leq \eta \|\nabla f(\mathbf{u}_0)\|$.

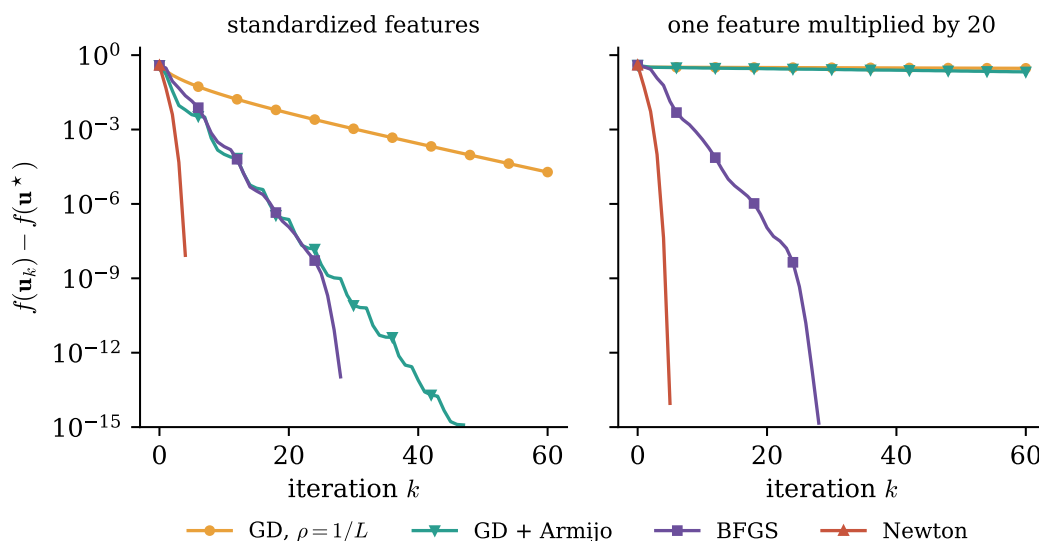


Figure 28: Convergence of gradient descent (with constant step $1/L$, or with Armijo’s backtracking), BFGS and Newton’s method (both with Armijo’s backtracking) on an ℓ_2 -regularized logistic regression ($m = 300$, $d = 5$), starting from $\mathbf{u}_0 = \mathbf{0}$; log scale on the vertical axis. Left: standardized features. Right: the same data with one feature multiplied by 20; the condition number increases, gradient descent becomes very slow, while Newton’s method and BFGS are almost insensitive to this change of scale.

[code: make_figures_ch02.py: fig_ch02_rates]

Remark large-scale learning In Machine Learning, the objective is usually an empirical risk $f(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \ell_i(\boldsymbol{\theta})$, where ℓ_i is the loss on the i -th example. When m is very large, even one evaluation of the full gradient is expensive, and Newton-type methods are out of reach when the number of parameters d is large. The workhorse is then *stochastic gradient descent*, which replaces ∇f by the gradient $\nabla \ell_i$ of a single example (or of a small batch) drawn at random (Robbins and Monro, 1951); it is presented with its variants (momentum, Adam) in Section 9.6. The survey of Bottou et al. (2018) explains why such “inexact” methods are well suited to learning.

Historical note Gradient descent goes back to Cauchy (1847), who proposed it to solve systems of equations arising in astronomy. Newton’s method was introduced by Newton and Raphson in the seventeenth century to find the roots of polynomials; applied to the equation $\nabla f(\mathbf{u}) = \mathbf{0}$, it becomes the optimization method presented here. The BFGS update was discovered independently, in 1970, by Broyden, Fletcher, Goldfarb and Shanno, which explains its name. The textbooks of Nocedal and Wright (2006) and Nesterov (2018) are excellent references for this section.

To remember from Section 3

- At a local minimum of a differentiable function, the gradient vanishes (Euler’s equation); a critical point is a local minimum if the Hessian is positive definite there, and every critical point of a convex function is a global min-

imum.

- Gradient descent $\mathbf{u}_{k+1} = \mathbf{u}_k - \rho_k \nabla f(\mathbf{u}_k)$ moves in the direction of steepest descent. With a constant step, it requires $\rho < 2/L$ (the classical choice is $\rho = 1/L$); otherwise the step is chosen by an exact or an approximate (Armijo) line search.
- Its speed is governed by the condition number $\kappa = L/\alpha$: the convergence is linear for strongly convex functions, and slow when the level sets are elongated.
- Newton's method uses the Hessian, is insensitive to the conditioning and converges quadratically near the solution, but each iteration costs $\mathcal{O}(d^3)$ and it must be damped far from the solution. Quasi-Newton methods (BFGS, L-BFGS) build an approximation of the inverse Hessian from gradients only.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

Optimality Conditions

- Q3.1.** Is a point where the gradient vanishes always a local minimum? Give a counterexample in dimension 2.
- Q3.2.** Classify the critical point $\mathbf{0}$ of $f(\mathbf{x}) = x_1^2 - 4x_1x_2 + x_2^2$.
- Q3.3.** The functions x^4 , $-x^4$ and x^3 all satisfy $f'(0) = f''(0) = 0$. What does this tell you about the second-order test?
- Q3.4.** True or false: if a differentiable function has a unique critical point, this point is its global minimum.
- Q3.5.** Why is every critical point of a differentiable convex function a global minimum?

Minimization Problems

- Q3.6.** When does the least squares problem $\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$ have a unique solution?
- Q3.7.** True or false: logistic regression has a closed-form solution, like linear regression.

Synthesis of the section

- Q3.8.** Why is $-\nabla f(\mathbf{u})$ called the direction of steepest descent?
- Q3.9.** What happens with a constant step $\rho = 0.3$ on $f(\mathbf{u}) = \frac{1}{2}(u_1^2 + 10u_2^2)$?
- Q3.10.** What is the best constant step for $f(\mathbf{u}) = \frac{1}{2}(u_1^2 + 10u_2^2)$, and the corresponding contraction factor?
- Q3.11.** Why are two successive directions of gradient descent with optimal step orthogonal?
- Q3.12.** Why must the constant c of Armijo's condition be smaller than 1?
- Q3.13.** How many iterations does Newton's method need on a strictly convex quadratic function? And gradient descent?
- Q3.14.** True or false: for a strictly convex function, the pure Newton iteration (step 1) converges from any starting point.
- Q3.15.** We replace f by $100f$. Which of gradient descent with fixed step and Newton's method is affected?

Q3.16. Why is BFGS preferred to Newton's method when d is large?

4 Constrained and Smooth Convex Optimization

In the previous section, the variable $\mathbf{x}$ could take any value in $\mathbb{R}^d$. In many learning problems, however, the parameters we look for must satisfy some **constraints**. This section explains how the optimality conditions of Section 3 must be adapted in this case. It introduces the Lagrangian, the dual problem and the Karush–Kuhn–Tucker (KKT) conditions, which are the key tools to understand support vector machines, and it presents a simple algorithm to solve constrained problems, projected gradient descent. Our presentation follows [Boyd and Vandenberghe \(2004\)](#), to which we refer the reader for further details and for the proofs that are omitted here.

About this section

Guiding question. *How do we minimize a loss when the parameters are not free, e.g. when they must have a bounded norm or form a probability distribution?*

In this section you will learn to:

- write a constrained problem in standard form and recognize a convex problem;
- interpret Lagrange multipliers geometrically, and build the Lagrangian and the dual problem;
- write and use the KKT conditions to solve a problem in closed form, and interpret complementary slackness (support vectors, active constraints);
- compute simple projections (ball, box, simplex) and run projected gradient descent.

4.1 Motivation: Constraints in Machine Learning

Constraints appear naturally in Machine Learning. Let us give three examples that will be met again later in the course.

- **Support vector machines.** Given a sample $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ with labels $y_i \in \{-1, +1\}$, the hard-margin SVM looks for the separating hyperplane $\{\mathbf{x} \mid \mathbf{w}^T \mathbf{x} + b = 0\}$ with the largest margin. It is obtained by solving

$$\min_{\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}} \frac{1}{2} \|\mathbf{w}\|_2^2 \quad \text{s.t.} \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1, \dots, m.$$

Each example gives an *inequality constraint*: it must lie on the right side of the hyperplane, at a distance at least $1/\|\mathbf{w}\|_2$.

- **Norm constraints and regularization.** To avoid overfitting, we often restrict the size of the parameters, e.g. by minimizing an empirical risk $F(\boldsymbol{\theta})$ under the constraint $\|\boldsymbol{\theta}\| \leq r$ (the ridge regression with $\|\cdot\|_2$, the Lasso with $\|\cdot\|_1$). We will see that such a constrained problem is closely related to the penalized problem $\min_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2$: the regularization parameter λ plays the role of a *Lagrange multiplier*.
- **Weights of a vote.** In ensemble methods (majority votes, boosting, PAC-Bayes), we learn a weight q_j for each voter h_j , $j = 1, \dots, n$. These weights must define a probability distribution, i.e. belong to the **simplex**

$$\Delta_n = \left\{ \mathbf{q} \in \mathbb{R}^n \mid q_j \geq 0, \sum_{j=1}^n q_j = 1 \right\}.$$

In all these cases, we cannot simply set the gradient of the objective function equal to zero: its minimizer over $\mathbb{R}^d$ is in general *not* feasible.

4.2 Problem Formulation

Definition 4.1: Constrained Optimization Problem

Let $f, g_1, \dots, g_p, h_1, \dots, h_q : \mathbb{R}^d \rightarrow \mathbb{R}$ be given functions. A **constrained optimization problem** (in standard form) is a problem of the form

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^d} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, p, \\ & h_j(\mathbf{x}) = 0, \quad j = 1, \dots, q. \end{aligned} \quad (4.1)$$

The function f is the **objective function**, the g_i 's define the **inequality constraints** and the h_j 's the **equality constraints**. The set

$$\mathcal{C} = \left\{ \mathbf{x} \in \mathbb{R}^d \mid g_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, p, \quad h_j(\mathbf{x}) = 0, \quad j = 1, \dots, q \right\}$$

is called the **feasible set**; its elements are the **feasible points**. The optimal value of the problem is denoted by $p^* = \inf_{\mathbf{x} \in \mathcal{C}} f(\mathbf{x})$, and a feasible point $\mathbf{x}^*$ such that $f(\mathbf{x}^*) = p^*$ is a **solution** (or minimizer) of the problem.

The problem is said to be **convex** if f and $g_1, \dots, g_p$ are convex functions and $h_1, \dots, h_q$ are affine functions, *i.e.* $h_j(\mathbf{x}) = \mathbf{a}_j^T \mathbf{x} - b_j$.

Note that any constraint can be written in this form: a constraint $g(\mathbf{x}) \geq 0$ becomes $-g(\mathbf{x}) \leq 0$, and maximizing f amounts to minimizing $-f$. For instance, the SVM constraints read $g_i(\mathbf{w}, b) = 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b) \leq 0$, and the simplex constraints read $-q_j \leq 0$ and $\sum_j q_j - 1 = 0$.

Why do we ask the equality constraints to be affine in a convex problem? Because the set $\{\mathbf{x} \mid h(\mathbf{x}) = 0\}$ is in general not convex when h is only convex (think of the circle $\{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_2^2 - 1 = 0\}$), whereas the following result holds.

Proposition 4.1: Convex Problems

For a convex problem (4.1):

- the feasible set $\mathcal{C}$ is convex;
- every local minimum of f on $\mathcal{C}$ is a global minimum;
- if f is differentiable, a feasible point $\mathbf{x}^*$ is a solution if and only if

$$\langle \nabla f(\mathbf{x}^*), \mathbf{x} - \mathbf{x}^* \rangle \geq 0, \quad \forall \mathbf{x} \in \mathcal{C}. \quad (4.2)$$

Proof. Each sublevel set $\{\mathbf{x} \mid g_i(\mathbf{x}) \leq 0\}$ of a convex function is convex (if $g_i(\mathbf{u}) \leq 0$ and $g_i(\mathbf{v}) \leq 0$, then $g_i(t\mathbf{u} + (1-t)\mathbf{v}) \leq tg_i(\mathbf{u}) + (1-t)g_i(\mathbf{v}) \leq 0$), and each set $\{\mathbf{x} \mid h_j(\mathbf{x}) = 0\}$ is a hyperplane, hence convex. $\mathcal{C}$ is an intersection of convex sets, so it is convex.

Let $\mathbf{x}^*$ be a local minimum and assume that there is $\mathbf{y} \in \mathcal{C}$ with $f(\mathbf{y}) < f(\mathbf{x}^*)$. For $t \in]0, 1]$, the point $\mathbf{x}_t = \mathbf{x}^* + t(\mathbf{y} - \mathbf{x}^*)$ belongs to $\mathcal{C}$ and, by convexity, $f(\mathbf{x}_t) \leq (1 - t)f(\mathbf{x}^*) + tf(\mathbf{y}) < f(\mathbf{x}^*)$. Since $\mathbf{x}_t$ is arbitrarily close to $\mathbf{x}^*$ when $t \rightarrow 0$, this contradicts the local optimality of $\mathbf{x}^*$.

Finally, condition (4.2) is necessary by Euler's inequality (Proposition 3.1), and it is sufficient because, by Proposition 2.6, $f(\mathbf{x}) \geq f(\mathbf{x}^*) + \langle \nabla f(\mathbf{x}^*), \mathbf{x} - \mathbf{x}^* \rangle \geq f(\mathbf{x}^*)$ for all $\mathbf{x} \in \mathcal{C}$. $\square$

Condition (4.2) has a nice geometric interpretation: at a solution, no feasible direction $\mathbf{x} - \mathbf{x}^*$ is a descent direction. When $\mathbf{x}^*$ lies in the interior of $\mathcal{C}$, all the directions are feasible and we recover Euler's equation $\nabla f(\mathbf{x}^*) = \mathbf{0}$. When $\mathbf{x}^*$ lies on the boundary of $\mathcal{C}$, the gradient does not vanish in general, but $-\nabla f(\mathbf{x}^*)$ must "point outside" the feasible set.

Geometric intuition Consider first a single equality constraint $h(\mathbf{x}) = 0$, as in the left part of Figure 29. Let us move along the curve $\{h = 0\}$ starting from a point $\mathbf{x}$. If $\nabla f(\mathbf{x})$ has a nonzero component along the curve (*i.e.* along its tangent), then we can decrease f while remaining feasible, so $\mathbf{x}$ is not optimal. At a solution $\mathbf{x}^*$, the gradient $\nabla f(\mathbf{x}^*)$ must therefore be **orthogonal to the feasible set**, *i.e.* parallel to $\nabla h(\mathbf{x}^*)$, which is itself orthogonal to the curve $\{h = 0\}$:

$$\nabla f(\mathbf{x}^*) + \mu \nabla h(\mathbf{x}^*) = \mathbf{0} \quad \text{for some } \mu \in \mathbb{R}.$$

Equivalently, the level set of f going through $\mathbf{x}^*$ is tangent to the curve $\{h = 0\}$.

For an inequality constraint $g(\mathbf{x}) \leq 0$ (right part of Figure 29), two situations may occur. Either the solution lies inside the feasible set, where $g(\mathbf{x}^*) < 0$: the constraint is *inactive* and $\nabla f(\mathbf{x}^*) = \mathbf{0}$. Or it lies on its boundary, where $g(\mathbf{x}^*) = 0$: the constraint is *active*, and $-\nabla f(\mathbf{x}^*)$ must point outwards, *i.e.* in the same direction as $\nabla g(\mathbf{x}^*)$:

$$\nabla f(\mathbf{x}^*) + \lambda \nabla g(\mathbf{x}^*) = \mathbf{0} \quad \text{for some } \lambda \geq 0.$$

The sign of the multiplier λ now matters: if $-\nabla f(\mathbf{x}^*)$ pointed inwards, we could decrease f by moving inside the feasible set.

4.3 Equality Constraints and Lagrange Multipliers

The geometric intuition above is made rigorous by the following classical theorem, which we state without proof (it relies on the implicit function theorem).

Theorem 4.2: Lagrange Multipliers

Let $f, h_1, \dots, h_q : \mathbb{R}^d \rightarrow \mathbb{R}$ be continuously differentiable functions, and let $\mathbf{x}^*$ be a local minimum of f on the set $\{\mathbf{x} \mid h_j(\mathbf{x}) = 0, j = 1, \dots, q\}$. Assume that the gradients $\nabla h_1(\mathbf{x}^*), \dots, \nabla h_q(\mathbf{x}^*)$ are linearly independent. Then there exist real numbers $\mu_1^*, \dots, \mu_q^*$, called **Lagrange multipliers**, such that

$$\nabla f(\mathbf{x}^*) + \sum_{j=1}^q \mu_j^* \nabla h_j(\mathbf{x}^*) = \mathbf{0}.$$

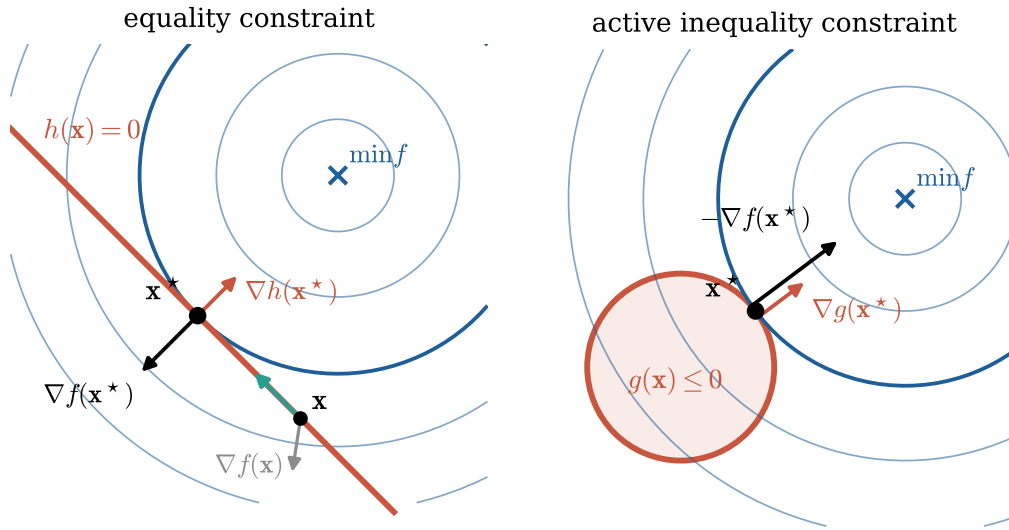


Figure 29: Geometry of constrained optimality. Left: with an equality constraint $h(\mathbf{x}) = 0$ (red line), the solution $\mathbf{x}^*$ is the point where a level set of f (blue circles) is tangent to the constraint; there, $\nabla f(\mathbf{x}^*)$ and $\nabla h(\mathbf{x}^*)$ are parallel. At the non-optimal point $\mathbf{x}$, $\nabla f(\mathbf{x})$ has a component along the line, and moving along the green direction decreases f while remaining feasible. Right: with an active inequality constraint $g(\mathbf{x}) \leq 0$ (red disk), the solution $\mathbf{x}^*$ lies on the boundary, and $-\nabla f(\mathbf{x}^*)$ points outside the feasible set, in the direction of $\nabla g(\mathbf{x}^*)$. The thick blue circle is the level set of f going through $\mathbf{x}^*$. [code: make_figures_ch02.py: fig_ch03_lagrange]

This condition can be written in a compact way by introducing the **Lagrangian**

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) = f(\mathbf{x}) + \sum_{j=1}^q \mu_j h_j(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^d, \quad \boldsymbol{\mu} \in \mathbb{R}^q.$$

The conditions of Theorem 4.2, together with the feasibility of $\mathbf{x}^*$, simply read

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}^*) = \mathbf{0} \quad \text{and} \quad \nabla_{\boldsymbol{\mu}} \mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}^*) = (h_1(\mathbf{x}^*), \dots, h_q(\mathbf{x}^*)) = \mathbf{0} :$$

we have replaced a constrained problem with d unknowns by an *unconstrained* system of $d + q$ equations with $d + q$ unknowns. For a convex problem, these conditions are also sufficient (this is a special case of Theorem 4.5 below).

Example 4.1 (Minimum-norm solution of a linear equation). Let $\mathbf{a} \in \mathbb{R}^d \setminus \{\mathbf{0}\}$ and $b \in \mathbb{R}$. We look for the point of the hyperplane $\{\mathbf{x} \mid \mathbf{a}^T \mathbf{x} = b\}$ that is closest to the origin:

$$\min_{\mathbf{x} \in \mathbb{R}^d} \frac{1}{2} \|\mathbf{x}\|_2^2 \quad \text{s.t.} \quad \mathbf{a}^T \mathbf{x} - b = 0.$$

The Lagrangian is $\mathcal{L}(\mathbf{x}, \mu) = \frac{1}{2} \|\mathbf{x}\|_2^2 + \mu(\mathbf{a}^T \mathbf{x} - b)$, and the optimality conditions give

$$\begin{aligned} \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \mu) = \mathbf{x} + \mu \mathbf{a} = \mathbf{0} &\iff \mathbf{x} = -\mu \mathbf{a} \\ &\downarrow \text{plug into the constraint } \mathbf{a}^T \mathbf{x} = b \\ -\mu \|\mathbf{a}\|_2^2 = b &\iff \mu^* = -\frac{b}{\|\mathbf{a}\|_2^2}. \end{aligned}$$

Hence $\mathbf{x}^* = \frac{b}{\|\mathbf{a}\|_2^2} \mathbf{a}$: the solution is colinear to the normal vector $\mathbf{a}$ of the hyperplane, and the distance from the origin to the hyperplane is $\|\mathbf{x}^*\|_2 = |b| / \|\mathbf{a}\|_2$. This is exactly the formula used to compute the margin of a linear classifier.

Example 4.2 (Quadratic function under linear constraints). Let $\mathbf{A} \in \mathbb{R}^{d \times d}$ be symmetric positive definite, $\mathbf{b} \in \mathbb{R}^d$, $\mathbf{C} \in \mathbb{R}^{q \times d}$ of rank q and $\mathbf{d} \in \mathbb{R}^q$. We consider

$$\min_{\mathbf{x} \in \mathbb{R}^d} \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} \quad \text{s.t.} \quad \mathbf{C} \mathbf{x} = \mathbf{d}.$$

With $\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + \boldsymbol{\mu}^T (\mathbf{C} \mathbf{x} - \mathbf{d})$, the optimality conditions form the linear system

$$\begin{pmatrix} \mathbf{A} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{d} \end{pmatrix}.$$

The first block gives $\mathbf{x} = \mathbf{A}^{-1}(\mathbf{b} - \mathbf{C}^T \boldsymbol{\mu})$; plugging it into $\mathbf{C} \mathbf{x} = \mathbf{d}$ gives $\mathbf{C} \mathbf{A}^{-1} \mathbf{b} - \mathbf{C} \mathbf{A}^{-1} \mathbf{C}^T \boldsymbol{\mu} = \mathbf{d}$. Since $\mathbf{C} \mathbf{A}^{-1} \mathbf{C}^T$ is invertible (it is positive definite because $\mathbf{C}$ has full rank),

$$\boldsymbol{\mu}^* = (\mathbf{C} \mathbf{A}^{-1} \mathbf{C}^T)^{-1} (\mathbf{C} \mathbf{A}^{-1} \mathbf{b} - \mathbf{d}) \quad \text{and} \quad \mathbf{x}^* = \mathbf{A}^{-1} (\mathbf{b} - \mathbf{C}^T \boldsymbol{\mu}^*).$$

Without constraints, we would have found $\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$: the term $-\mathbf{A}^{-1} \mathbf{C}^T \boldsymbol{\mu}^*$ is the correction due to the constraints.

4.4 Lagrangian Duality

We now come back to the general problem (4.1), with inequality constraints, which we call the **primal problem**. The idea of duality is to incorporate the constraints into the objective function, with a price for violating them.

Definition 4.2: Lagrangian and Dual Function

The **Lagrangian** associated with problem (4.1) is the function $\mathcal{L} : \mathbb{R}^d \times \mathbb{R}^p \times \mathbb{R}^q \rightarrow \mathbb{R}$ defined by

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) + \sum_{i=1}^p \lambda_i g_i(\mathbf{x}) + \sum_{j=1}^q \mu_j h_j(\mathbf{x}).$$

The variables $\boldsymbol{\lambda} = (\lambda_1, \dots, \lambda_p)$ and $\boldsymbol{\mu} = (\mu_1, \dots, \mu_q)$ are the **Lagrange multipliers** (or **dual variables**); the multipliers λ_i of the inequality constraints are required to be nonnegative. The **dual function** is defined, for $\boldsymbol{\lambda} \in \mathbb{R}^p$ with $\boldsymbol{\lambda} \geq \mathbf{0}$ and $\boldsymbol{\mu} \in \mathbb{R}^q$, by

$$\mathcal{G}(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{\mathbf{x} \in \mathbb{R}^d} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \in \mathbb{R} \cup \{-\infty\}.$$

For each fixed $\mathbf{x}$, the function $(\boldsymbol{\lambda}, \boldsymbol{\mu}) \mapsto \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ is affine. As a pointwise infimum of affine functions, **the dual function $\mathcal{G}$ is always concave**, even when the primal problem is not convex (by the same argument as for the maximum of convex functions in Proposition 2.5).

Definition 4.3: Dual Problem

The **dual problem** of (4.1) is

$$\max_{\lambda \in \mathbb{R}^p, \mu \in \mathbb{R}^q} \mathcal{G}(\lambda, \mu) \quad \text{s.t.} \quad \lambda \geq \mathbf{0}. \quad (4.3)$$

Its optimal value is denoted by d^* . A pair (λ, μ) with $\lambda \geq \mathbf{0}$ and $\mathcal{G}(\lambda, \mu) > -\infty$ is said to be **dual feasible**.

Since $\mathcal{G}$ is concave and the constraint $\lambda \geq \mathbf{0}$ is convex, the dual problem is always a convex problem (maximizing a concave function amounts to minimizing a convex one). The fundamental property of the dual problem is that it provides lower bounds on the optimal value of the primal problem.

Theorem 4.3: Weak Duality

For every $\lambda \geq \mathbf{0}$ and every $\mu \in \mathbb{R}^q$, we have $\mathcal{G}(\lambda, \mu) \leq p^*$. Consequently,

$$d^* \leq p^*.$$

Proof. Let $\mathbf{x}$ be a feasible point and $\lambda \geq \mathbf{0}$, $\mu \in \mathbb{R}^q$. Then

$$\begin{aligned}
\mathcal{G}(\lambda, \mu) &= \inf_{\mathbf{z} \in \mathbb{R}^d} \mathcal{L}(\mathbf{z}, \lambda, \mu) \\
&\downarrow \text{the infimum is smaller than any value of the function} \\
&\leq \mathcal{L}(\mathbf{x}, \lambda, \mu) = f(\mathbf{x}) + \sum_{i=1}^p \lambda_i g_i(\mathbf{x}) + \sum_{j=1}^q \underbrace{\mu_j h_j(\mathbf{x})}_{=0} \\
&\downarrow \lambda_i \geq 0 \text{ and } g_i(\mathbf{x}) \leq 0, \text{ so each term } \lambda_i g_i(\mathbf{x}) \text{ is nonpositive} \\
&\leq f(\mathbf{x}).
\end{aligned}$$

Taking the infimum over all feasible points $\mathbf{x}$ gives $\mathcal{G}(\lambda, \mu) \leq p^*$, and taking the supremum over (λ, μ) gives $d^* \leq p^*$. $\square$

The difference $p^* - d^* \geq 0$ is called the **duality gap**. Weak duality always holds, but the gap may be positive, in particular for nonconvex problems. When $d^* = p^*$, we say that **strong duality** holds: solving the dual problem, which is often easier, then gives the optimal value of the primal problem. For convex problems, strong duality holds under mild assumptions called *constraint qualifications*; the most popular one is Slater's condition.

Theorem 4.4: Strong Duality – Slater's Condition

Assume that problem (4.1) is convex and that there exists a **strictly feasible** point $\tilde{\mathbf{x}}$, i.e.

$$g_i(\tilde{\mathbf{x}}) < 0, \quad i = 1, \dots, p, \quad \text{and} \quad h_j(\tilde{\mathbf{x}}) = 0, \quad j = 1, \dots, q.$$

Then strong duality holds, $d^* = p^*$, and if $p^* > -\infty$, the dual problem has a solution (λ^*, μ^*) . Moreover, strict feasibility is not required for the inequality

constraints g_i that are affine: in particular, if all the constraints are affine, strong duality holds as soon as the problem is feasible.

The proof relies on a separating hyperplane argument; see [Boyd and Vandenberghe \(2004, Section 5.3.2\)](#).

Example 4.3 (Dual of the minimum-norm problem). Let us come back to the problem $\min \frac{1}{2} \|\mathbf{x}\|_2^2$ s.t. $\mathbf{a}^T \mathbf{x} = b$. For a fixed μ , the Lagrangian $\mathcal{L}(\mathbf{x}, \mu) = \frac{1}{2} \|\mathbf{x}\|_2^2 + \mu(\mathbf{a}^T \mathbf{x} - b)$ is a convex quadratic function of $\mathbf{x}$, minimized at $\mathbf{x} = -\mu \mathbf{a}$. Hence

$$\mathcal{G}(\mu) = \frac{1}{2} \mu^2 \|\mathbf{a}\|_2^2 - \mu^2 \|\mathbf{a}\|_2^2 - \mu b = -\frac{1}{2} \mu^2 \|\mathbf{a}\|_2^2 - \mu b.$$

This concave function of μ is maximized at $\mu^* = -b / \|\mathbf{a}\|_2^2$, and $d^* = \mathcal{G}(\mu^*) = \frac{b^2}{2 \|\mathbf{a}\|_2^2}$, which is exactly $p^* = \frac{1}{2} \|\mathbf{x}^*\|_2^2$: strong duality holds, as predicted by Theorem 4.4 (the constraint is affine).

4.5 The Karush–Kuhn–Tucker Conditions

We can now state the optimality conditions of constrained problems, which generalize Euler's equation $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and Lagrange's theorem. They were obtained by Karush (1939) and, independently, by Kuhn and Tucker (1951).

Theorem 4.5: Karush–Kuhn–Tucker (KKT) Conditions

Assume that $f, g_1, \dots, g_p, h_1, \dots, h_q$ are differentiable. We say that $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ satisfies the **KKT conditions** if

1. **(stationarity)** $\nabla f(\mathbf{x}^*) + \sum_{i=1}^p \lambda_i^* \nabla g_i(\mathbf{x}^*) + \sum_{j=1}^q \mu_j^* \nabla h_j(\mathbf{x}^*) = \mathbf{0}$, i.e. $\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = \mathbf{0}$;
2. **(primal feasibility)** $g_i(\mathbf{x}^*) \leq 0$ for all i and $h_j(\mathbf{x}^*) = 0$ for all j ;
3. **(dual feasibility)** $\lambda_i^* \geq 0$ for all i ;
4. **(complementary slackness)** $\lambda_i^* g_i(\mathbf{x}^*) = 0$ for all i .

Then:

- **necessity:** if strong duality holds, $\mathbf{x}^*$ is a solution of the primal problem (4.1) and $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ is a solution of the dual problem (4.3), then $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ satisfies the KKT conditions;
- **sufficiency:** if the problem is convex and $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ satisfies the KKT conditions, then $\mathbf{x}^*$ is a solution of the primal problem, $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ is a solution of the dual problem, and $p^* = d^*$.

In particular, for a convex problem satisfying Slater's condition, $\mathbf{x}^*$ is a solution if and only if there exist $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ such that the KKT conditions hold.

Proof. Necessity. Since $\mathbf{x}^*$ is feasible and $\boldsymbol{\lambda}^* \geq \mathbf{0}$, primal and dual feasibility hold. Moreover,

$$\begin{aligned} f(\mathbf{x}^*) &= p^* = d^* = \mathcal{G}(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = \inf_{\mathbf{x} \in \mathbb{R}^d} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \\ &\leq \mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = f(\mathbf{x}^*) + \sum_{i=1}^p \lambda_i^* g_i(\mathbf{x}^*) + \sum_{j=1}^q \underbrace{\mu_j^* h_j(\mathbf{x}^*)}_{=0} \\ &\quad \downarrow \text{ each } \lambda_i^* g_i(\mathbf{x}^*) \text{ is nonpositive} \\ &\leq f(\mathbf{x}^*). \end{aligned}$$

Hence all these inequalities are equalities. The first one says that $\mathbf{x}^*$ minimizes the differentiable function $\mathbf{x} \mapsto \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ over $\mathbb{R}^d$, so its gradient vanishes at $\mathbf{x}^*$ by Euler's equation (Proposition 3.2): this is stationarity. The second one says that $\sum_i \lambda_i^* g_i(\mathbf{x}^*) = 0$; since each term of this sum is nonpositive, each term is zero: this is complementary slackness.

Sufficiency. Since the problem is convex and $\boldsymbol{\lambda}^* \geq \mathbf{0}$, the function $\mathbf{x} \mapsto \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ is convex (a nonnegative combination of convex functions plus affine functions). By stationarity and Proposition 2.6, $\mathbf{x}^*$ is a global minimizer of this function, hence

$$\mathcal{G}(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = \mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = f(\mathbf{x}^*) + \sum_{i=1}^p \underbrace{\lambda_i^* g_i(\mathbf{x}^*)}_{=0} + \sum_{j=1}^q \underbrace{\mu_j^* h_j(\mathbf{x}^*)}_{=0} = f(\mathbf{x}^*),$$

using complementary slackness and primal feasibility. By weak duality, $\mathcal{G}(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \leq d^* \leq p^* \leq f(\mathbf{x}^*)$, so all these quantities are equal: $\mathbf{x}^*$ is primal optimal, $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ is dual optimal and $p^* = d^*$. $\square$

Remark Without a constraint qualification, the KKT conditions may fail to hold at a solution, even for a convex problem. Consider $\min_{x \in \mathbb{R}} x$ s.t. $g(x) = x^2 \leq 0$. The only feasible point, hence the solution, is $x^* = 0$, but stationarity would require $1 + 2\lambda x^* = 1 = 0$, which is impossible. Here Slater's condition fails: no point satisfies $x^2 < 0$.

Complementary slackness and its interpretation Complementary slackness says that, for each inequality constraint,

$$\lambda_i^* > 0 \implies g_i(\mathbf{x}^*) = 0 \quad \text{and} \quad g_i(\mathbf{x}^*) < 0 \implies \lambda_i^* = 0.$$

In words: a constraint that is not **active** at the solution ($g_i(\mathbf{x}^*) < 0$) does not play any role, and its multiplier is zero; the solution would not change if this constraint were removed. Conversely, a positive multiplier indicates an active constraint, which really “pushes” the solution. This is what we observed in Figure 29: an inactive constraint gives $\nabla f(\mathbf{x}^*) = \mathbf{0}$, an active one gives $-\nabla f(\mathbf{x}^*) = \lambda^* \nabla g(\mathbf{x}^*)$ with $\lambda^* \geq 0$.

The multipliers also have an economic interpretation, as **prices** or **sensitivities**: if the i -th constraint is relaxed into $g_i(\mathbf{x}) \leq u_i$ with u_i small, then, under regularity assumptions, the optimal value changes by approximately $-\lambda_i^* u_i$. A large multiplier thus indicates a constraint which is expensive, *i.e.* which strongly limits the decrease of the objective (Boyd and Vandenberghe, 2004, Section 5.6).

In the SVM, the multipliers are attached to the training examples: complementary slackness implies that only the examples whose constraint is active, the **support vectors**, have a nonzero multiplier and contribute to the solution (see the preview below).

4.6 Worked Examples

The KKT conditions often allow us to compute the solution in closed form, by examining which constraints are active.

Example 4.4 (Projection onto a Euclidean ball). Let $\mathbf{z} \in \mathbb{R}^d$ and $r > 0$. We look for the point of the ball $\{\mathbf{x} \mid \|\mathbf{x}\|_2 \leq r\}$ closest to $\mathbf{z}$:

$$\min_{\mathbf{x} \in \mathbb{R}^d} \frac{1}{2} \|\mathbf{x} - \mathbf{z}\|_2^2 \quad \text{s.t.} \quad g(\mathbf{x}) = \frac{1}{2} (\|\mathbf{x}\|_2^2 - r^2) \leq 0.$$

The problem is convex and Slater's condition holds ($g(\mathbf{0}) < 0$), so the KKT conditions characterize the solution:

$$\mathbf{x} - \mathbf{z} + \lambda \mathbf{x} = \mathbf{0}, \quad \|\mathbf{x}\|_2 \leq r, \quad \lambda \geq 0, \quad \lambda (\|\mathbf{x}\|_2^2 - r^2) = 0.$$

Stationarity gives $\mathbf{x} = \mathbf{z}/(1 + \lambda)$. Two cases occur.

- If $\|\mathbf{z}\|_2 \leq r$, then $\lambda = 0$ and $\mathbf{x} = \mathbf{z}$ satisfy all the conditions: the constraint is inactive and $\mathbf{z}$ is its own projection.
- If $\|\mathbf{z}\|_2 > r$, then $\lambda = 0$ is impossible (it would give $\mathbf{x} = \mathbf{z}$, which is not feasible), so the constraint is active: $\|\mathbf{x}\|_2 = r$, *i.e.* $1 + \lambda = \|\mathbf{z}\|_2 / r$, and $\mathbf{x} = r \mathbf{z} / \|\mathbf{z}\|_2$.

In both cases, the projection is

$$\Pi_{\mathcal{B}(r)}(\mathbf{z}) = \min \left(1, \frac{r}{\|\mathbf{z}\|_2} \right) \mathbf{z} :$$

the point $\mathbf{z}$ is simply rescaled onto the sphere when it lies outside the ball. This operation is known as *gradient clipping* in deep learning.

Example 4.5 (Projection onto the simplex). Let $\mathbf{z} \in \mathbb{R}^n$. We look for the probability vector closest to $\mathbf{z}$:

$$\min_{\mathbf{q} \in \mathbb{R}^n} \frac{1}{2} \|\mathbf{q} - \mathbf{z}\|_2^2 \quad \text{s.t.} \quad -q_j \leq 0, \quad j = 1, \dots, n, \quad \sum_{j=1}^n q_j - 1 = 0.$$

All the constraints are affine, so the KKT conditions are necessary and sufficient. With multipliers $\lambda_j \geq 0$ for the inequalities and μ for the equality, they read

$$q_j - z_j - \lambda_j + \mu = 0, \quad q_j \geq 0, \quad \lambda_j \geq 0, \quad \lambda_j q_j = 0, \quad \sum_{j=1}^n q_j = 1.$$

Let us set $\tau = \mu$ and discuss according to complementary slackness:

- if $q_j > 0$, then $\lambda_j = 0$ and $q_j = z_j - \tau$ (which requires $z_j > \tau$);
- if $q_j = 0$, then $\lambda_j = \tau - z_j \geq 0$, *i.e.* $z_j \leq \tau$.

In both cases, $q_j = \max(z_j - \tau, 0)$, where the threshold τ is determined by the constraint

$$\sum_{j=1}^n \max(z_j - \tau, 0) = 1.$$

The left-hand side is a continuous and nonincreasing function of τ , decreasing from $+\infty$ to 0 on the range where it is positive, so this equation has a unique solution, which can be found by sorting the z_j 's. For instance, with $\mathbf{z} = (0.8, 0.6, -0.2)$, we try to keep the two largest components: $(0.8 - \tau) + (0.6 - \tau) = 1$ gives $\tau = 0.2$, and indeed $z_3 = -0.2 \leq \tau$. Hence $\mathbf{q}^* = (0.6, 0.4, 0)$: the projection onto the simplex produces *sparse* weights.

Example 4.6 (A preview of the hard-margin SVM dual). Let us apply Lagrangian duality to the hard-margin SVM problem of the motivation, with the constraints $g_i(\mathbf{w}, b) = 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b) \leq 0$. With multipliers $\alpha_i \geq 0$, the Lagrangian is

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{w}\|_2^2 + \sum_{i=1}^m \alpha_i (1 - y_i(\mathbf{w}^T \mathbf{x}_i + b)).$$

For a fixed $\boldsymbol{\alpha}$, this function is convex in $(\mathbf{w}, b)$, and it is minimized where its gradient vanishes:

$$\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{w} - \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i = \mathbf{0} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b} = - \sum_{i=1}^m \alpha_i y_i = 0.$$

If $\sum_i \alpha_i y_i \neq 0$, the Lagrangian is affine and nonconstant in b , so $\mathcal{G}(\boldsymbol{\alpha}) = -\infty$. Otherwise, plugging $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$ into $\mathcal{L}$ gives the dual problem

$$\max_{\boldsymbol{\alpha} \in \mathbb{R}^m} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{k=1}^m \alpha_i \alpha_k y_i y_k \mathbf{x}_i^T \mathbf{x}_k \quad \text{s.t.} \quad \alpha_i \geq 0, \quad \sum_{i=1}^m \alpha_i y_i = 0.$$

Since the constraints are affine, strong duality holds as soon as the data are linearly separable. Three important facts can already be observed:

- the solution $\mathbf{w}^* = \sum_i \alpha_i^* y_i \mathbf{x}_i$ is a linear combination of the training examples;
- by complementary slackness, $\alpha_i^* > 0$ only if $y_i(\mathbf{w}^{*T} \mathbf{x}_i + b^*) = 1$, *i.e.* only for the examples lying exactly on the margin: the **support vectors**;
- the data only appear through the inner products $\mathbf{x}_i^T \mathbf{x}_k$, which opens the way to *kernels*.

These properties are studied in detail in the section devoted to support vector machines in Part II.

4.7 Projected Gradient Descent

The KKT conditions do not always lead to a closed-form solution, and we need algorithms. When the feasible set $\mathcal{C}$ is simple, the most natural idea is to perform a gradient step, which may leave $\mathcal{C}$, and then to come back to $\mathcal{C}$ by projecting onto it.

Algorithm 1: Projected Gradient Descent

Input: a differentiable function f , a closed convex set $\mathcal{C}$ with an easy projection $\Pi_{\mathcal{C}}$, an initial point $\mathbf{x}_0 \in \mathcal{C}$, a step size $\rho > 0$, a number of iterations K (or a tolerance $\eta > 0$)

Output: an approximate solution $\mathbf{x}_K$ of $\min_{\mathbf{x} \in \mathcal{C}} f(\mathbf{x})$

```
1 for  $k = 0, 1, \dots, K - 1$  do
2    $\mathbf{y}_{k+1} \leftarrow \mathbf{x}_k - \rho \nabla f(\mathbf{x}_k)$  ; // gradient step
3    $\mathbf{x}_{k+1} \leftarrow \Pi_{\mathcal{C}}(\mathbf{y}_{k+1})$  ; // projection onto the feasible set
4   if  $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2 \leq \rho \eta$  then
5     stop and return  $\mathbf{x}_{k+1}$ 
6 return  $\mathbf{x}_K$ 
```

Definition 4.4: Projection onto a Convex Set

Let $\mathcal{C} \subset \mathbb{R}^d$ be a nonempty, closed and convex set. The **projection** of a point $\mathbf{z} \in \mathbb{R}^d$ onto $\mathcal{C}$ is

$$\Pi_{\mathcal{C}}(\mathbf{z}) = \arg \min_{\mathbf{x} \in \mathcal{C}} \frac{1}{2} \|\mathbf{x} - \mathbf{z}\|_2^2.$$

Proposition 4.6: Properties of the Projection

Let $\mathcal{C}$ be a nonempty, closed and convex subset of $\mathbb{R}^d$.

- For all $\mathbf{z} \in \mathbb{R}^d$, the projection $\Pi_{\mathcal{C}}(\mathbf{z})$ exists and is unique.
- A point $\mathbf{p} \in \mathcal{C}$ is the projection of $\mathbf{z}$ if and only if

$$\langle \mathbf{z} - \mathbf{p}, \mathbf{x} - \mathbf{p} \rangle \leq 0, \quad \forall \mathbf{x} \in \mathcal{C}.$$

- The projection is non-expansive: $\|\Pi_{\mathcal{C}}(\mathbf{z}) - \Pi_{\mathcal{C}}(\mathbf{z}')\|_2 \leq \|\mathbf{z} - \mathbf{z}'\|_2$ for all $\mathbf{z}, \mathbf{z}'$.

Proof (of the second point). The function $\mathbf{x} \mapsto \frac{1}{2} \|\mathbf{x} - \mathbf{z}\|_2^2$ is convex, with gradient $\mathbf{x} - \mathbf{z}$. By Proposition 4.1 (or directly by Euler's inequality and Proposition 2.6), $\mathbf{p}$ minimizes it over $\mathcal{C}$ if and only if $\langle \mathbf{p} - \mathbf{z}, \mathbf{x} - \mathbf{p} \rangle \geq 0$ for all $\mathbf{x} \in \mathcal{C}$. Existence follows from the continuity of the function and the fact that we can restrict the search to the compact set $\mathcal{C} \cap \{\mathbf{x} \mid \|\mathbf{x} - \mathbf{z}\|_2 \leq \|\mathbf{x}_0 - \mathbf{z}\|_2\}$ for any $\mathbf{x}_0 \in \mathcal{C}$, and uniqueness from the strong convexity of the function. $\square$

Geometrically, the angle between $\mathbf{z} - \Pi_{\mathcal{C}}(\mathbf{z})$ and any vector pointing from $\Pi_{\mathcal{C}}(\mathbf{z})$ into $\mathcal{C}$ is obtuse. Combining this characterization with condition (4.2), one checks that, for any $\rho > 0$, $\mathbf{x}^*$ is a solution of $\min_{\mathbf{x} \in \mathcal{C}} f(\mathbf{x})$ (with f convex and differentiable) if and only if

$$\mathbf{x}^* = \Pi_{\mathcal{C}}(\mathbf{x}^* - \rho \nabla f(\mathbf{x}^*)).$$

The solutions are thus the fixed points of the map $\mathbf{x} \mapsto \Pi_{\mathcal{C}}(\mathbf{x} - \rho \nabla f(\mathbf{x}))$, which suggests the following algorithm.

Note that the stopping criterion is not $\|\nabla f(\mathbf{x}_k)\|_2 \leq \eta$ anymore, since the gradient does not vanish at a solution lying on the boundary of $\mathcal{C}$; we use instead the distance between two successive iterates, which is zero exactly at the fixed points.

The projections computed in the previous examples can directly be plugged into Algorithm 1: onto a ball, the projection is a rescaling; onto a box $[a, b]^d$, it amounts to clipping each coordinate, $x_j \mapsto \min(\max(x_j, a), b)$; onto the simplex, it is the thresholding operation computed above. Figure 30 shows the algorithm at work on a ball and on a box. As long as the iterates remain inside $\mathcal{C}$, the algorithm coincides with gradient descent; once the boundary is reached, each gradient step leaves $\mathcal{C}$ and the projection brings the iterate back, so that the iterates slide along the boundary towards the point where $-\nabla f$ is orthogonal to it, as in Figure 29.

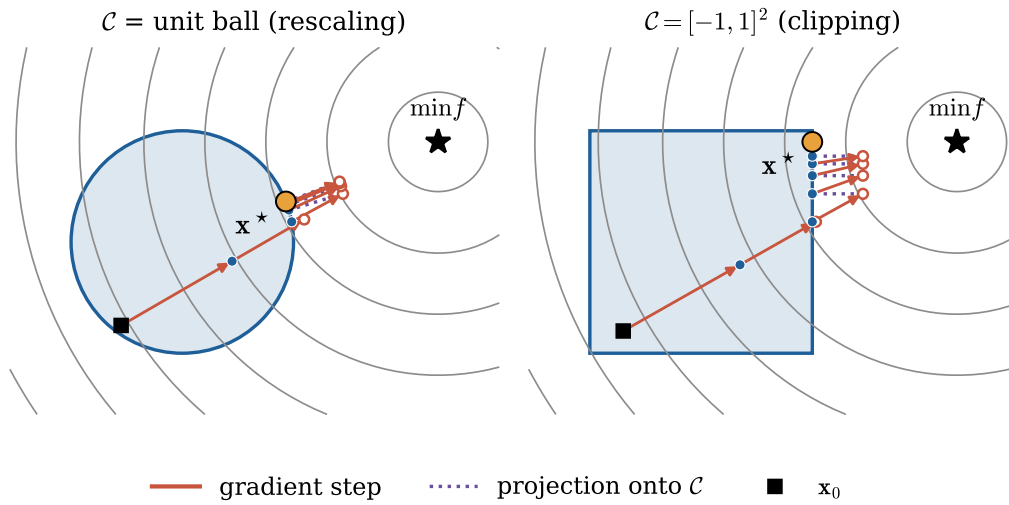


Figure 30: Six iterations of projected gradient descent (step $\rho = 0.35$) for a quadratic function whose unconstrained minimizer (star) is not feasible, starting from the black square. Each gradient step (red arrow) may leave the feasible set $\mathcal{C}$ (blue); the projection (dotted line) brings the point back onto $\mathcal{C}$. Left: unit ball, the projection is a rescaling. Right: box $[-1, 1]^2$, the projection clips each coordinate. In both cases, the constraint is active at the solution $\mathbf{x}^*$, where the gradient does not vanish.

[code: make_figures_ch02.py: fig_ch03_projgd]

Remark on convergence Projected gradient descent enjoys the same guarantees as gradient descent in the unconstrained case. Assume that f is convex and L -smooth, *i.e.* that its gradient is L -Lipschitz: $\|\nabla f(\mathbf{u}) - \nabla f(\mathbf{v})\|_2 \leq L \|\mathbf{u} - \mathbf{v}\|_2$. With the step $\rho = 1/L$, the iterates satisfy

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{L \|\mathbf{x}_0 - \mathbf{x}^*\|_2^2}{2k},$$

a convergence rate in $\mathcal{O}(1/k)$. If f is moreover α -strongly convex, the convergence is linear (geometric): $\|\mathbf{x}_k - \mathbf{x}^*\|_2^2 \leq (1 - \alpha/L)^k \|\mathbf{x}_0 - \mathbf{x}^*\|_2^2$. The projection does not slow down the algorithm, but each iteration is only cheap if the projection is cheap: for complicated feasible sets, computing $\Pi_{\mathcal{C}}$ is itself an optimization problem, and one rather uses duality or other methods (interior point methods, Frank–Wolfe algorithm, ...).

To remember from Section 4

- A constrained problem is convex when the objective and the inequality constraints are convex and the equality constraints are affine; then every local minimum is global.
- At a solution, $-\nabla f(\mathbf{x}^*)$ is a nonnegative combination of the gradients of the active constraints (plus any combination of those of the equality constraints): this is the stationarity condition of the Lagrangian.
- The dual function is always concave and gives lower bounds on the optimal value (weak duality); for convex problems satisfying Slater's condition, the duality gap is zero (strong duality).
- The KKT conditions (stationarity, primal and dual feasibility, complementary slackness) characterize the solutions of convex problems satisfying Slater's condition. Complementary slackness says that inactive constraints have zero multipliers, which explains the support vectors of the SVM.
- When the projection onto $\mathcal{C}$ is cheap (ball, box, simplex), projected gradient descent alternates a gradient step and a projection, with the same convergence guarantees as gradient descent.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

Equality Constraints and Lagrange Multipliers

- Q4.1.** Why must the equality constraints of a convex problem be affine?
- Q4.2.** Is the problem $\min \|\mathbf{x}\|_2^2$ s.t. $\|\mathbf{x}\|_2 \geq 1$ convex? What are its solutions?
- Q4.3.** Find the point of the line $x_1 + x_2 = 2$ closest to the origin, and the Lagrange multiplier of $\min \frac{1}{2} \|\mathbf{x}\|_2^2$ s.t. $x_1 + x_2 - 2 = 0$.
- Q4.4.** True or false: at the solution of an equality-constrained problem, $\nabla f(\mathbf{x}^*) = \mathbf{0}$.

Worked Examples

- Q4.5.** Why is the dual problem always convex, even when the primal problem is not?
- Q4.6.** A feasible point gives $f(\mathbf{x}) = 3.2$ and a dual feasible pair gives $\mathcal{G}(\boldsymbol{\lambda}, \boldsymbol{\mu}) = 3.0$. What can you say?
- Q4.7.** True or false: for a convex problem, the KKT conditions always hold at a solution.
- Q4.8.** A multiplier satisfies $\lambda_i^* > 0$. What can you say about the i -th constraint? And if $g_i(\mathbf{x}^*) < 0$?
- Q4.9.** What is the projection of $\mathbf{z} = (2, 0.5, 0)$ onto the simplex Δ_3 ?

Synthesis of the section

- Q4.10.** What is the projection of $\mathbf{z} = (3, 4)$ onto the unit ball? onto the box $[-1, 1]^2$?
- Q4.11.** Project $\mathbf{z} = (1.5, -0.3)$ onto the box $[0, 1]^2$.
- Q4.12.** Why can we not use $\|\nabla f(\mathbf{x}_k)\|_2 \leq \eta$ as a stopping criterion in projected gradient descent?

- Q4.13.** True or false: if the unconstrained minimizer of f belongs to $\mathcal{C}$, projected gradient descent produces the same iterates as gradient descent.
- Q4.14.** Why does the fixed-point characterization $\mathbf{x}^* = \Pi_{\mathcal{C}}(\mathbf{x}^* - \rho \nabla f(\mathbf{x}^*))$ hold for every $\rho > 0$?
- Q4.15.** What is the cost of one iteration of projected gradient descent on the simplex Δ_n ?

Part II

Supervised Machine Learning

This part introduces the supervised learning problem and the main algorithms used to solve it. [Section 5](#) describes the data and the different learning tasks; [Section 6](#) presents the framework of statistical learning theory, which explains what it means for a model to *generalize*, through empirical risk minimization and generalization bounds; [Section 7](#) details the learning procedure in practice: how to train a model, tune its hyperparameters and evaluate it with performance measures adapted to the task. [Section 8](#) then presents the classical supervised algorithms — the k -nearest neighbours, support vector machines, linear and logistic regression, kernel regression and decision trees — and [Section 9](#) is devoted to neural networks. For each algorithm we describe the model, the optimization problem it solves, and its strengths and weaknesses in practice.

5 Introduction

In this first section, we present some generalities on what machine learning is, on the importance of data, and on the various problems related to finding a good representation of these data.

About this section

Guiding question. *What does it mean for a machine to “learn”, and what do we need to make it learn?*

In this section you will learn to:

- *state Mitchell’s definition of learning (task T , performance P , experience E) and recognize these three ingredients in a concrete application;*
- *explain why every type of data (images, tables, texts) must eventually be turned into a vector of features;*
- *describe the main steps of a learning pipeline, from raw data to predictions;*
- *distinguish supervised, unsupervised and reinforcement learning, and, within supervised learning, regression from classification.*

5.1 Generalities

Machine Learning is a subfield of *Artificial Intelligence* at the frontier of computer science and *Applied Mathematics* (statistics and optimization). This discipline also partially overlaps with Data Science, as it relies on collecting data that are analyzed and studied in order to extract the relevant information for the application at hand.

The main question we would like to answer is:

Can machines think?

In other words, are machines able to perform tasks that are usually done by humans (A. Turing), such as:

- driving a car,
- creating a masterpiece,
- recognizing people,
- detecting anomalies in medical images,
- etc.

Tom Mitchell **Mitchell** (1997) has provided a more formal definition of *Machine Learning*²:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .

²The book is available here: <https://www.cs.cmu.edu/~tom/mlbook.html>. Its first chapters may be of interest to the reader.

The experience E can be thought of as a collection of data, but also as the training process used to train your *algorithm*. Generally speaking, the more (good) data we have and the better we train our model, the better the results will be; we will see, however, that training “too much” on the same data can also be harmful. In any case, everything depends on the performance measure P that is used, and we will come back to this point later. The task T can be, for instance, a classification or a regression task, as studied in the linear models course.

In a nutshell Let us identify the three ingredients on a spam filter. The task T is to decide whether an incoming email is a spam or not; the performance P is, for instance, the proportion of emails that are correctly sorted; the experience E is a collection of emails that the users have already flagged as spam or as legitimate. The program *learns* if its proportion of correctly sorted emails increases as more flagged emails become available. Note that nobody writes the rules (“if the email contains the word *lottery*, then...”) by hand: the rules are *inferred from the data*.

Historical note The question “Can machines think?” is the opening sentence of the famous paper of Alan Turing [Turing \(1950\)](#), in which he proposes the *imitation game* (now called the Turing test) and already suggests that, rather than programming an adult mind, one could build a “child machine” and *educate* it. The expression *machine learning* itself was popularized a few years later by Arthur Samuel [Samuel \(1959\)](#), whose checkers-playing program improved by playing against itself, a very early instance of learning from experience.

To sum up, to solve a given task, and *to help our machine think or solve a problem*, we need a collection of observations, *i.e.* a collection of data.

5.2 Data and Machine Learning problems

Another definition of Machine Learning can be stated as follows:

Machine Learning is a field of Computer Science, and a subfield of Artificial Intelligence, which aims to design and study algorithms that are able to learn from data and to make predictions.

Depending on the application, the data can be of many different natures: images, videos, raw data, categorical data, trees, graphs, time series, etc.

A concrete example is illustrated in Figure [31](#) with two images. The aim of the algorithm is to distinguish dogs from cats in a collection of images. In this case, the algorithm is trained to solve a classification problem where the output can take two values: *dog* or *cat*.

We also often deal with data that can be represented in a table. This is the case in the following regression task, where we aim to predict the score obtained in a language test (y) from the scores obtained in four other tests, x_1, x_2, x_3 and x_4 (see Table [1](#)).

Although these data may look very different in their usual representation, a machine necessarily sees them in the same form. This form is most often the one of the



Figure 31: An example of data that can be used for classification tasks. These two images are extracted from the *Dogs vs Cats* dataset, available on [Kaggle](#).

y	x_1	x_2	x_3	x_4
125	13	18	25	11
158	39	18	59	30
207	52	50	62	53
182	29	43	50	29
196	50	37	65	56

Table 1: Data describing the score obtained in a language test according to the scores obtained in four other tests.

second example, *i.e.* an array of values, in other words a matrix.

This data representation is one of the important pillars of machine learning. Although it may seem simple at first, obtaining a good representation of the data is far from trivial. It is even an important research topic, very often guided by the problem we are trying to solve. Representation learning is particularly important in *Natural Language Processing* (NLP), where one needs to learn a good representation of words or sentences in the form of *vectors*.

Let us go back to our images to see this more precisely. Remember that an image can be seen as a collection of *pixels* (or *voxels* if you are working in a three-dimensional space), each of which is associated with a number. Generally, in a black and white (grayscale) image, the pixel value lies between 0 and 255 (or between 0 and 1 if the data are *scaled*). A color image, such as the ones presented in Figure 31, is represented by three different matrices³, where each entry represents the pixel intensity associated with a color channel (R: red, G: green and B: blue):

$$R = \begin{pmatrix} 0.1 & 1 & 0.23 \\ 0.12 & 0.23 & 0.95 \\ 0 & 0.45 & 0.22 \end{pmatrix}, \quad G = \begin{pmatrix} 0.34 & 1 & 0.19 \\ 0.67 & 0.53 & 0.95 \\ 1 & 0.95 & 0.23 \end{pmatrix}, \quad B = \begin{pmatrix} 0.56 & 0.83 & 0.23 \\ 0.74 & 0.81 & 0.34 \\ 0.45 & 0.45 & 0.29 \end{pmatrix}.$$

However, these raw features have some flaws:

³In practice, images are not left in matrix form: we use models, such as convolutional neural networks, to learn a vector representation of these images; such vectors are then easier to manipulate by subsequent algorithms.

- the intensity of the pixels depends on the brightness, but also on the conditions under which the image was captured;
- the shape of the object in the image depends on the viewing angle, and the object may be harder to distinguish from its background;
- finally, the apparent size of the object depends on its distance from the camera.

For all these reasons, it is important to define (or find) descriptors that are independent of such environmental factors; this is the objective of a *Representation Learning* course. These methods are generally based on neural network algorithms, which we will present later in this document.

As we can see, data are at the core of Machine Learning algorithms (they can be compared to the experience E mentioned by Tom Mitchell). We have also seen that they can take different forms: they can be raw, or transformed to make them usable by an algorithm (*e.g.* the translation of a text or a word into a vector of numbers). Finally, the descriptors of these data can also be learned in order to obtain a representation of the data suited to the task at hand.

A question one might then ask when observing a series of numbers is:

Do the observed values of the different descriptors follow a well-known distribution?

The observed values do follow some distribution. However, and this is the main problem we have to deal with in Machine Learning, this distribution **is always unknown**. We will see that this fact has consequences on the learning process of an algorithm and on its performance.

Let us first focus on the use of the data and go back to the image classification task with *cats* and *dogs*. Suppose that we have a collection of such images, as illustrated in Figure 32 (keep in mind that each image is actually a point in a d -dimensional space, where d is the length of the vector representing it, *i.e.* the number of descriptors of the image), and that we want to classify these images, *i.e.* to learn an algorithm, or a model, which is able to predict the class of an image and thus to separate the two types of images.

To build such an algorithm, we need to define three things:

- **Input:** the data that will be used for the task, *e.g.* our images of cats and dogs (in vector form);
- **Output:** the answer to the task under study, *e.g.* the label of the image;
- **Model:** a collection of objects that are used to solve the problem (objective function, type of hypothesis and optimization process).

These ingredients fit together in the *pipeline* summarized in Figure 33, which will serve as a map for the rest of this document: Section 6 formalizes the training phase, Section 7 explains how to evaluate the resulting model, and the following sections present several learning algorithms.

In this case, we could learn several types of separators, or classifiers, for the images, such as straight lines or more complex curves in the feature space. The classifier we learn will try to minimize the classification error on the observed data, *i.e.* we will try to minimize the number of images of dogs classified as cats and vice versa. This will be done based on the descriptors of the images as well as on their labels. Through a



Figure 32: A collection of images that can be used for a classification task. These images are extracted from the *Dogs vs Cats* dataset, available on [Kaggle](#).

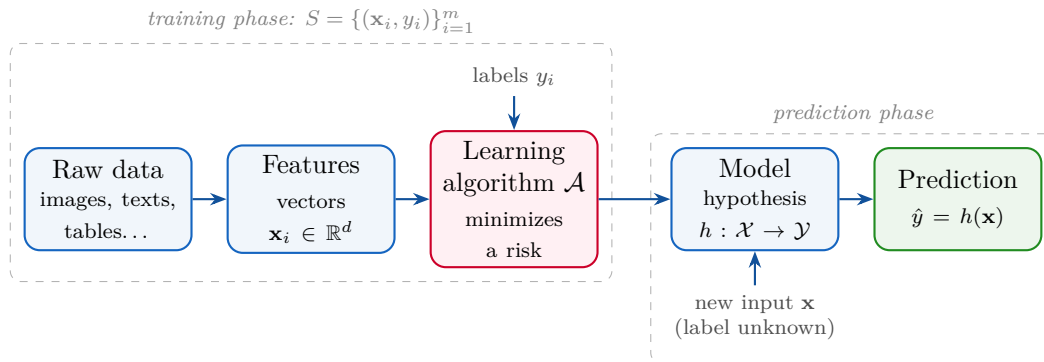


Figure 33: The supervised learning pipeline. Raw data are first turned into feature vectors; a learning algorithm then uses these vectors together with their labels to select a model h , which is finally applied to new inputs whose label is unknown. Choosing the representation (first arrow) is often as important as choosing the algorithm.

learning process that we will detail later, the algorithm will then iteratively learn from its errors in order to improve and reach an optimal solution. Once this model is learned, it is applied to new data (*e.g.* new images) that it has to classify without knowing their labels.

Pitfall Learning makes sense only if the new data “look like” the training data, *i.e.* if they are generated by the same (unknown) distribution. A cat-versus-dog classifier trained on daylight pictures may fail on night pictures, and a fraud detector trained on last year’s transactions may become obsolete when fraudsters change their habits. This *distribution shift* is one of the main causes of failure of machine learning systems in practice; the theory presented in the next section relies on the assumption that training and test examples are drawn independently

from the same distribution.

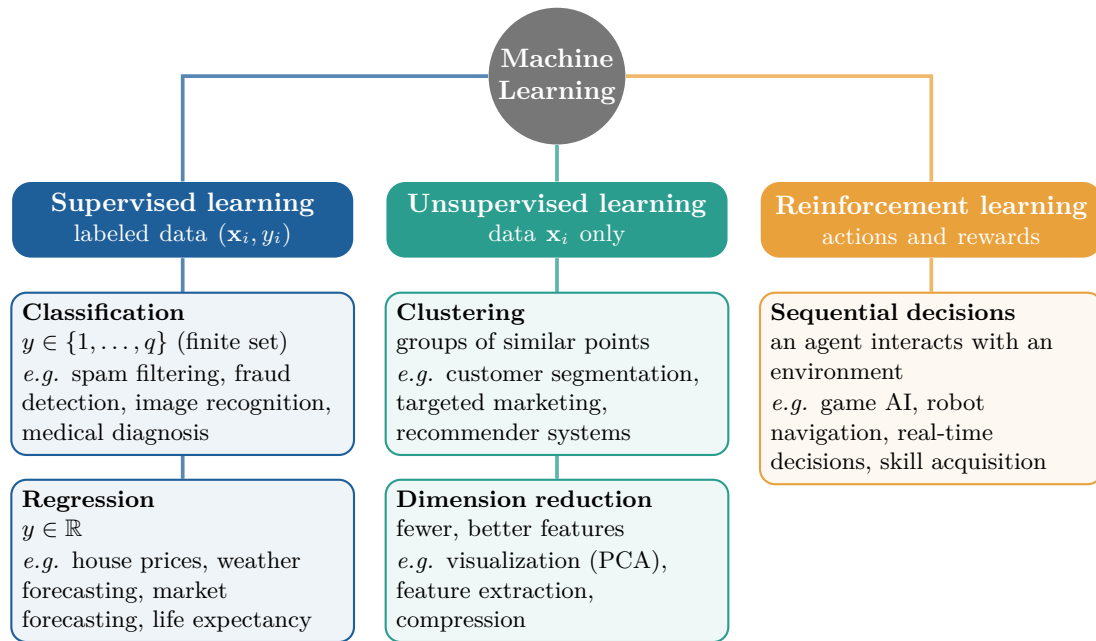


Figure 34: The three main branches of Machine Learning, the main tasks within each of them and some typical applications. Supervised learning, on the left, is the subject of this course. (Adapted from a diagram found [here](#).)

As we have seen, once the data are collected (and often completed and cleaned), they can be used for several machine learning tasks, such as regression, *e.g.* to predict the price of a share, or the price of a house according to its characteristics. They can also be exploited for classification tasks, *e.g.* to discriminate spam from ham (legitimate email) when receiving an email, to identify whether a transaction is fraudulent or genuine, or to detect anomalies in a medical examination such as a blood test. In both cases, the data are labeled by a variable y : it is a real number in regression and a category in classification, as illustrated in Figure 35.

When we have a genuine transaction or a ham email, the example is usually labeled -1 (it is called a negative example), while it is labeled 1 (positive example) when it is an object of interest, such as a spam or a fraudulent transaction. When such label information is used by a Machine Learning algorithm, we talk about *Supervised Learning*; otherwise, we talk about *Unsupervised Learning*⁴. The difference is illustrated in Figure 36: the data points are the same, but the questions we can ask about them are not.

There also exists another branch of Machine Learning called *Reinforcement Learning*⁵. For the sake of completeness, we give a brief description of each branch below; more information can be found in Figure 34.

Supervised Learning This branch includes all classification and regression tasks. It covers applications such as fraud detection, recommendation systems, time series

⁴This branch will be studied in a dedicated course.

⁵We do not study Reinforcement Learning here.

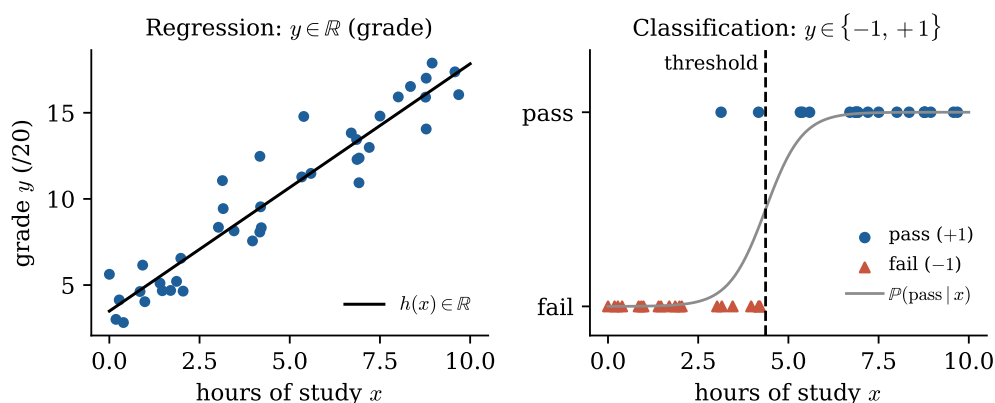


Figure 35: The same input (number of hours of study of a student) used for two different tasks. Left: *regression*, the output y is the grade, a real number, and the model h returns a real number. Right: *classification*, the output is pass/fail; a model (here a logistic regression) estimates the probability of passing, and thresholding it gives a classifier.

[code: make_figures_ch05.py:

fig_ch04_reg_classif]

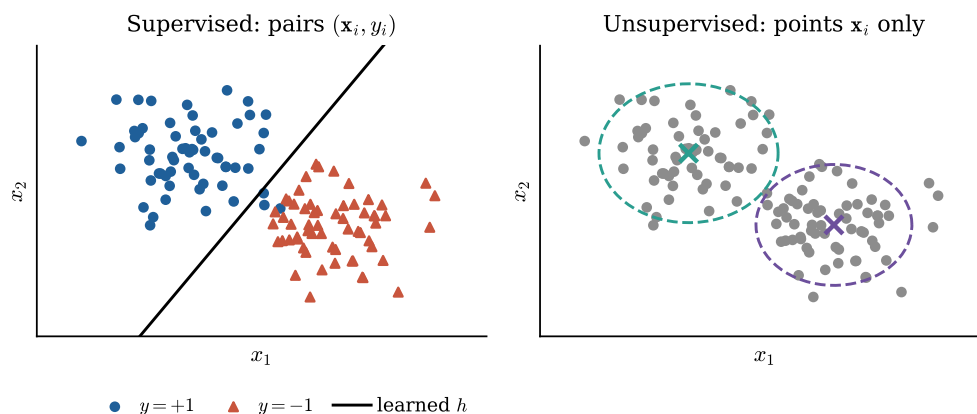


Figure 36: Supervised *versus* unsupervised learning on the same points. Left: each point comes with a label, and we learn a classifier h (here a straight line) that predicts the label of new points. Right: no label is available; an unsupervised method (here k -means with $k = 2$) looks for a structure in the data, *e.g.* groups of similar points, without any notion of “correct answer”.

[code: make_figures_ch05.py: fig_ch04_sup_unsup]

analysis, the study of medical images, EEG and ECG signals, the prediction of the number of people infected by a given disease or virus, or of the stock market value of a given company, etc. The important point here is the access to labeled data.

Some research fields, such as transfer learning (how to exploit the knowledge learned in one domain in order to apply it to a related domain), can also be classified in this branch. Many algorithms belong to this field. Readers who have taken a course on linear models will certainly remember the Gaussian linear model (regression) or logistic regression (classification). Other models are of course commonly used, and we will present them in the next sections.

Unsupervised Learning Unlike in supervised learning, the data used here have no label. The analysis is then based only on the distribution of the data. Applica-

tions include anomaly detection, clustering (*e.g.* population segmentation for marketing strategies), and recommendation systems based on customer typology. Another aspect is dimension reduction, via statistical methods such as Principal Component Analysis, and feature construction.

We can also include approaches such as unsupervised domain adaptation, which is the unsupervised counterpart of a subcategory of transfer learning.

Reinforcement Learning This last category differs from the first two in that there is no fixed dataset per se. Here, the system evolves in an environment and must be able to take the right actions according to this environment and to its own state, through a reward/punishment system.

A simple example is the automatic piloting of a drone whose goal is to avoid obstacles in order to reach its destination. The environment can then be considered as the set of objects that are close to the drone. The state of the drone can be its structural state as well as its position relative to the elements around it, and the actions correspond to the movements of the drone (left, right, up, down). When the drone hits or gets too close to an obstacle, we decrease the value of a quantity that reflects the reward obtained by the system.

In this course, we will focus on the first branch, *i.e.* on Supervised Learning. Our objective will be to learn a classifier h , also called a hypothesis, from a collection of labeled data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, in order to classify new instances, where each $\mathbf{x}_i$ is a vector of descriptors. The next section focuses on the theoretical aspects of Supervised Learning.

Remark In this document, the words *hypothesis*, *model*, *predictor* and *classifier* (or *regressor*) are used almost interchangeably to denote the function h returned by the learning algorithm. Similarly, *features*, *descriptors*, *variables* and *attributes* all denote the coordinates of $\mathbf{x}$, and *examples*, *instances*, *observations* or *data points* denote the pairs $(\mathbf{x}_i, y_i)$.

Requirements In order to understand the different notions presented in this document, and in Machine Learning in general, the reader is invited to review the fundamental concepts in *Probability/Statistics*, *Linear Models*, *Linear Algebra* (operations on matrices: determinant, transpose, multiplication, eigenvalues, eigenvectors) and finally *Analysis* (differentiation, integration, study of functions of several variables with real or vector values, sequences of functions and sequences of real vectors). Part I recalls the main tools of linear algebra and optimization that will be needed. For a gentle and non-technical overview of the practical lessons of machine learning, the reader may also enjoy Domingos (2012).

To remember from Section 5

- A program *learns* when its performance P on a task T improves with experience E , which in practice is a collection of data.
- Whatever their original nature (images, texts, tables), data are represented as vectors of features $\mathbf{x} \in \mathbb{R}^d$ and gathered in a matrix; finding a good representation is a problem in itself.

- The distribution of the data is always **unknown**: we only observe a finite sample, and new data are assumed to come from the same distribution.
- Supervised learning uses labeled data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ to learn a hypothesis h ; it is called regression when y is real-valued and classification when y takes finitely many values.
- Unsupervised learning works without labels (clustering, dimension reduction), and reinforcement learning learns actions from rewards.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

Generalities

- Q5.1.** Identify the task T , the performance measure P and the experience E of a program that plays chess.
- Q5.2.** True or false? “A program whose rules were all written by hand by experts learns, in the sense of Mitchell, as soon as it performs well.”
- Q5.3.** Does more experience E always improve the performance P ?
- Q5.4.** Why does the definition of learning depend on the choice of the performance measure P ?

Synthesis of the section

- Q5.5.** For a system that recommends movies, give a possible task T , performance measure P and experience E .
- Q5.6.** Is predicting tomorrow’s temperature a regression or a classification task? And predicting whether it will rain tomorrow?
- Q5.7.** A 64×64 color image is turned into a vector by stacking its pixel values. What is the dimension d of this vector?
- Q5.8.** You are given the customer database of a company, without any additional information, and you are asked to find groups of similar customers. Which branch of machine learning is this?
- Q5.9.** Why can we not simply compute the error of a model on the true distribution of the data?
- Q5.10.** True or false? “Since the labels y_i are observed, the distribution of the data is known in supervised learning.”
- Q5.11.** In the pipeline of [Figure 33](#), at which stage are the labels y_i used? Are labels needed to make a prediction for a new input?
- Q5.12.** Give two reasons why the raw pixel values of a photograph are poor features to recognize a cat.
- Q5.13.** A spam filter labels spam emails $+1$ and legitimate emails -1 . Does “positive” mean “good”?
- Q5.14.** True or false? “Principal Component Analysis is a supervised method, since it produces new variables.”

- Q5.15.** A fraud detector was trained on last year's transactions and its error rate has doubled this year, although the code has not changed. What is the most likely explanation?
- Q5.16.** What is the main difference between the data used in reinforcement learning and in supervised learning?

6 Statistical Learning Theory

In this section, we first introduce the notion of *risk*, the quantity we aim to minimize in most of the optimization problems arising in supervised learning. We will then see that it is possible to provide theoretical guarantees on the performance of the learned model using statistical tools such as concentration inequalities.

About this section

Guiding question. *A model makes few errors on the data it was trained on: what can we say about its errors on data it has never seen?*

In this section you will learn to:

- define the true risk and the empirical risk of a hypothesis, and explain why only the latter can be computed;
- describe empirical, structural and regularized risk minimization, and relate them to under- and over-fitting;
- decompose the error of a learned model into an approximation error and an estimation error;
- read a generalization bound: confidence $1 - \delta$, accuracy ε , sample size m and complexity of $\mathcal{H}$ (cardinality, VC-dimension) or stability of the algorithm.

Let us denote by $\mathcal{X}$ the input space of our data, also called the feature space, and by $\mathcal{Y}$ the output space⁶. The input space, *i.e.* the features used to describe the data, can be of different types: it can consist of continuous or discrete descriptors. In this document, we will focus on continuous features, *i.e.* $\mathcal{X} \subset \mathbb{R}^d$.

The output space can also vary from one task to another: it can be discrete, real-valued, or even consist of structured data. The nature of the output gives important information on the nature of the problem and leads the user to choose the appropriate tools.

- When $\mathcal{Y} \subset \mathbb{R}$ or $\mathcal{Y} = [0, 1]$, *i.e.* when the output is a real value, the aim is to learn a score or a probability, or to estimate a quantity, as is usually done in *regression tasks*.
- When $\mathcal{Y} = \{-1, +1\}$ or $\{0, 1\}$, *i.e.* when the output is binary, we aim to classify the data into two categories, also called *classes*. This type of task is called *binary classification*. Note that the output can take more than two values, *i.e.* $\mathcal{Y} = \{1, \dots, q\}$, where q is the number of classes. In this case, we talk about *multi-class classification*.

To solve these different tasks, we use an algorithm $\mathcal{A}$ which aims at learning a function $h : \mathcal{X} \rightarrow \mathcal{Y}$, called a *hypothesis*. From a practical point of view, the joint distribution $\mathcal{D}$ of the data over $\mathcal{X} \times \mathcal{Y}$ is unknown, and we only have access to a collection of *sample points*, *i.e.* a set of observations $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ drawn i.i.d. from $\mathcal{D}$. The goal is then to learn, from the sample S , a function h which is “good enough” on this training sample but which is also able to perform well on a new example $(\mathbf{x}, y)$.

The observed data are usually gathered in a matrix $\mathbf{X} \in \mathbb{R}^{m \times d}$ as follows:

⁶Note that there may be no output space, *i.e.* $\mathcal{Y} = \emptyset$, as in *unsupervised learning*.

$$\mathbf{X} = \begin{matrix} & \mathbf{v}_1 & \cdots & \mathbf{v}_k & \cdots & \mathbf{v}_d \\ \mathbf{x}_1 & \left(\begin{array}{ccccc} x_{11} & \cdots & x_{1k} & \cdots & x_{1d} \\ \vdots & & \vdots & & \vdots \\ \mathbf{x}_i & \begin{array}{ccccc} x_{i1} & \cdots & x_{ik} & \cdots & x_{id} \\ \vdots & & \vdots & & \vdots \\ \mathbf{x}_m & \begin{array}{ccccc} x_{m1} & \cdots & x_{mk} & \cdots & x_{md} \end{array} \end{array} \right), \end{matrix}$$

where each row represents an observation (m observations) and each column represents a variable, or descriptor (d columns).

In the next subsections, we explain how the function h is learned and what we mean by the performance of an algorithm.

6.1 Empirical Risk Minimization

In order to solve a given task, the algorithm $\mathcal{A}$ needs a criterion to optimize, called a *loss function* and usually denoted by $\ell(h(\cdot), \cdot) : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}_+$. In a regression task, it can be based on the difference between the estimated value $h(\mathbf{x})$ and the true value y , for example $\ell(h(\mathbf{x}), y) = |h(\mathbf{x}) - y|$ or $\ell(h(\mathbf{x}), y) = (h(\mathbf{x}) - y)^2$. In a classification task, the most natural way is to count the number of errors made by the classifier h . The corresponding loss function is then defined by $\ell(h(\mathbf{x}), y) = \mathbb{1}_{\{h(\mathbf{x}) \neq y\}}$ and is called the *0-1 loss*. However, this loss is neither convex nor differentiable (it is piecewise constant), and minimizing the empirical error it induces is known to be NP-hard in general, even over simple classes such as linear classifiers. That is why we usually replace the *0-1 loss* by a convex *surrogate* function, such as the hinge loss $\max(0, 1 - y h(\mathbf{x}))$ or the logistic loss $\ln(1 + e^{-y h(\mathbf{x})})$ when $\mathcal{Y} = \{-1, +1\}$ and h returns a real-valued score.

Whatever the hypothesis h , what we would really like to minimize is the expected value of its loss over the distribution $\mathcal{D}$, which leads to the *True Risk*.

Definition 6.1: True Risk

Let $\ell : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ be a loss function and $\mathcal{D}$ the distribution of the data. The *True Risk* $\mathcal{R}^\ell$ of a hypothesis h is defined by:

$$\mathcal{R}^\ell(h) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(h(\mathbf{x}), y)].$$

However, we cannot compute this quantity in practice because the joint distribution $\mathcal{D}$ of the data is unknown. Instead, we minimize the *Empirical Risk*, *i.e.* the average error over the sample S .

Definition 6.2: Empirical Risk

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a collection of m examples drawn i.i.d. from $\mathcal{D}$ and $\ell : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ a loss function. The *Empirical Risk* of a hypothesis h is defined by:

$$\mathcal{R}_S^\ell(h) = \frac{1}{m} \sum_{i=1}^m \ell(h(\mathbf{x}_i), y_i).$$

In other words, the empirical risk is the mean value of the loss over all the observed data.

Note that, for a *fixed* hypothesis h , $\mathcal{R}_S^\ell(h)$ is an unbiased estimator of $\mathcal{R}^\ell(h)$, since $\mathbb{E}_{S \sim \mathcal{D}^m}[\mathcal{R}_S^\ell(h)] = \mathcal{R}^\ell(h)$ by linearity of the expectation (see Section 1, where risks are introduced as expectations). We therefore minimize the empirical risk in the hope that the sample S is representative of the unknown distribution.

In a nutshell The true risk is the average error of h over *all* the examples the distribution $\mathcal{D}$ could generate, weighted by their probability; the empirical risk is the average error over the m examples we happened to observe. It is the same relation as between the mean of a population and the mean of a sample in a survey: the latter is a noisy, but computable, estimate of the former.

At first glance, minimizing the empirical risk might then seem equivalent to minimizing the true (or generalization) risk. This is not the case in general: the learned hypothesis depends on S , and we run the risk of *over-fitting* the data, as illustrated in Figure 37.

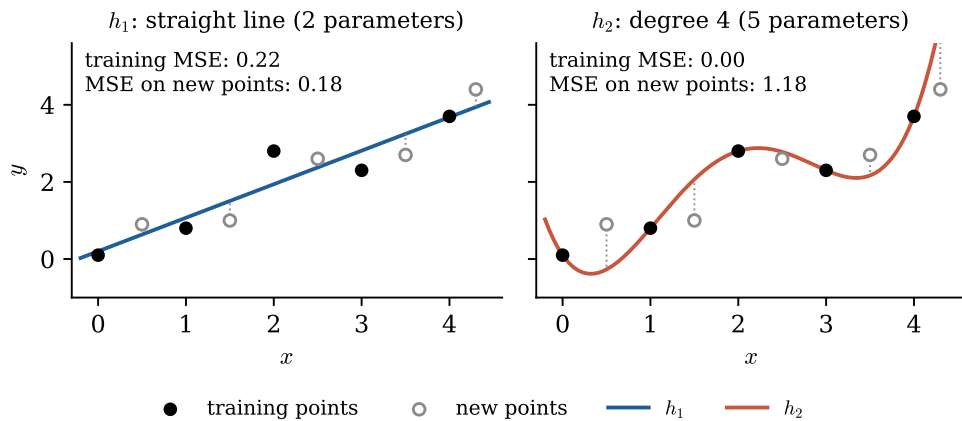


Figure 37: Illustration of the over-fitting phenomenon on a regression task with five training points (black). On the left, the learned hypothesis h_1 (a straight line) does not fit the training data exactly but generalizes well to new points drawn from the same distribution (white); on the right, the hypothesis h_2 , the polynomial of degree 4 interpolating the five points, has a zero training error but a much larger error on the new points (dotted segments). [code: make_figures_ch05.py: fig_ch05_overfit]

In this figure, we can see that the model learned on the right is more complex (it is a curve of degree 4) than the one on the left, which is a straight line. This remark shows that it is important to impose some constraints on the hypotheses we can learn in order to avoid such a situation.

Pitfall A small training error is **not** a proof that a model is good: the interpolating polynomial h_2 of Figure 37 has a training error equal to zero. The performance of a model must always be assessed on data that were not used to learn it (see Section 7), or certified by a generalization bound such as the ones presented in Section 6.2.

We present below a non-exhaustive list of ways to constrain the learned hypothesis.

Empirical risk minimization. The most straightforward solution is to fix in advance the hypothesis space $\mathcal{H}$, called a *set of hypotheses* or *function class*. Then, given this set of hypotheses $\mathcal{H}$, a sample S and a loss function ℓ , we look for the hypothesis h_S^* defined as a solution of:

$$h_S^* = \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{R}_S^\ell(h).$$

If the set $\mathcal{H}$ is small enough, we can avoid the situation previously described. However, without any side information on the data, it is hard to choose the appropriate set of hypotheses for the task at hand.

Remark (approximation and estimation errors) The choice of $\mathcal{H}$ is a trade-off, which can be made precise as follows. Let $\mathcal{R}^*$ be the smallest risk achievable by *any* function (the *Bayes risk*), and let $h_{\mathcal{H}}$ be the best hypothesis of $\mathcal{H}$, *i.e.* the minimizer of the true risk over $\mathcal{H}$ (assumed to exist). Then the excess risk of h_S^* splits into two terms:

$$\mathcal{R}^\ell(h_S^*) - \mathcal{R}^* = \underbrace{\mathcal{R}^\ell(h_S^*) - \mathcal{R}^\ell(h_{\mathcal{H}})}_{\text{estimation error}} + \underbrace{\mathcal{R}^\ell(h_{\mathcal{H}}) - \mathcal{R}^*}_{\text{approximation error}}.$$

The approximation error only depends on $\mathcal{H}$ and decreases when $\mathcal{H}$ grows; the estimation error comes from the fact that we only see a finite sample, and it typically increases with the size of $\mathcal{H}$ and decreases with m (see Figure 38). This decomposition is studied in more detail in Section 10.

Structural risk minimization. [See Section 4.1 of *Vapnik (1995)*] The idea here is to consider a growing sequence of sets of hypotheses $\{\mathcal{H}_n\}_{n \in \mathbb{N}}$, *i.e.* $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \dots$, so that the complexity (or the number of hypotheses) grows with n , and to choose the hypothesis h_S^* solution of the following minimization problem:

$$h_S^* = \operatorname{argmin}_{h \in \mathcal{H}_n, n \in \mathbb{N}} \mathcal{R}_S^\ell(h) + \operatorname{pen}(n, d).$$

Compared to the previous formulation, a *penalty term* is added to the empirical risk. This term measures the complexity (or size) of the set of hypotheses $\mathcal{H}_n$ and is an increasing function of n (note that it may also depend on the dimension d of the data). The aim is then to find a hypothesis with a low empirical risk in a set $\mathcal{H}_n$ that is as small as possible. With such a formulation, we are able to find a good trade-off between empirical risk minimization and the complexity of the learned model, *i.e.* to find a hypothesis that generalizes well. However, choosing a relevant sequence of sets of hypotheses remains hard in practice.

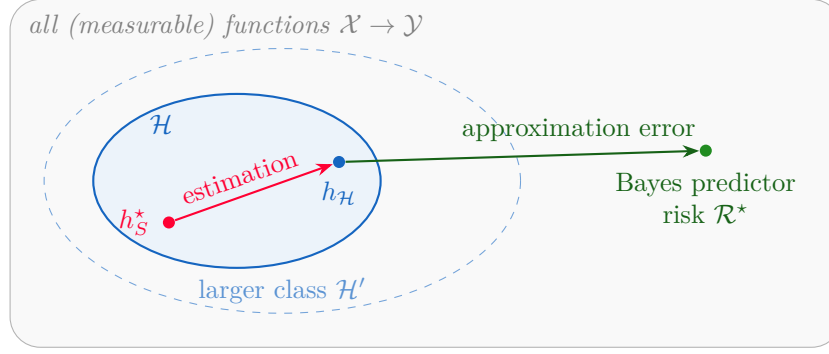


Figure 38: Decomposition of the excess risk of the hypothesis h_S^* learned on a sample S . The *approximation error* measures how far the best hypothesis h_H of the class is from the best possible predictor; the *estimation error* measures how far h_S^* is from h_H because of the finite sample. Enlarging the class (dashed ellipse) reduces the first error but makes the second one larger: this is the trade-off of Figure 39.

Regularized risk minimization. The preferred solution in practice, because it is easier to implement, is to choose a sufficiently large set of hypotheses $\mathcal{H}$ and to add a constraint on the parameters θ of the learned model $h = h_\theta$. The added term, $\lambda \|\theta\|$, is called a *regularization term*, and $\lambda \geq 0$ is the associated *regularization constant*:

$$h_S^* = \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{R}_S^\ell(h) + \lambda \|\theta\|.$$

Through the parameter λ , we are able to indirectly control the size, or complexity, of the set of hypotheses, for instance by constraining the norm of the parameters of the model⁷.

Example 6.1. Such a formulation was met in the Linear Models course when the lasso regression was presented. In this problem, we aim to find the best linear model, described by a vector θ (the intercept being included, hence the dimension $d + 1$), which fits a set of data $(\mathbf{y}, \mathbf{X})$. We also require the chosen model to be as sparse as possible, *i.e.* to be based on as few features as possible. To do so, we solve the following minimization problem:

$$\min_{\theta \in \mathbb{R}^{d+1}} \|\mathbf{y} - \mathbf{X}\theta\|_2^2 + \lambda \|\theta\|_1,$$

where $\|\cdot\|_1$ is the lasso regularization term, which induces a sparse solution for a sufficiently large λ .

In this last expression (as in the previous example), the smaller the value of λ , the more importance we give to fitting the data well. Conversely, the greater λ , the more importance we give to the complexity of the learned model: the errors made by the algorithm become insignificant compared to the control of the magnitude of the parameters θ . The risk is then that, when $\lambda \rightarrow \infty$, we face an *under-fitting* phenomenon: the model is too simple to capture the structure of the data and generalizes poorly.

⁷In the rest of the document, such a regularization constant will be called a *hyper-parameter* instead of a parameter, to distinguish it from the parameters of the learned hypothesis h .

The aim is thus to learn a model with an appropriate value of λ , *i.e.* to find a good trade-off between the minimization of the empirical risk and the complexity of the hypothesis, as depicted in Figure 39. To *tune* the value of λ , we typically use a *k-fold cross-validation* on the training set: we split the dataset into k folds of (roughly) equal size. This process will be described later; in short, it consists in learning a hypothesis (model) on $k - 1$ folds of the training data and testing its performance on the remaining one. The process is repeated k times, so that each fold is used once for testing, and we keep the value of λ that achieves the lowest average error on the held-out folds.

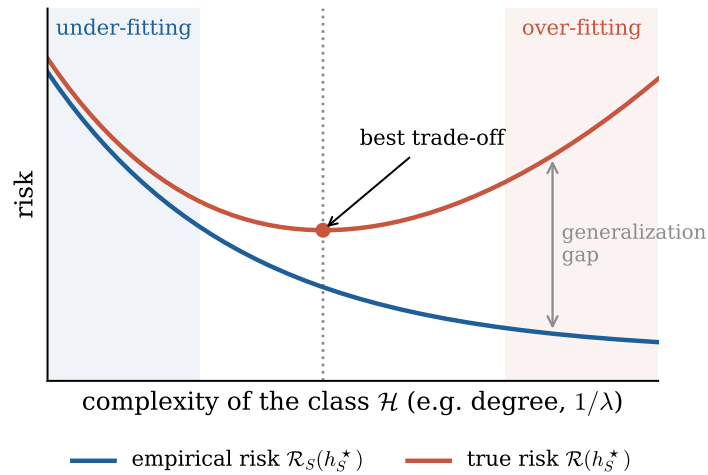


Figure 39: Evolution of the empirical and true risks of the learned hypothesis h_S^* according to the complexity of the set of hypotheses (schematic curves). The empirical risk always decreases with the complexity, whereas the true risk first decreases (less under-fitting) and then increases (over-fitting); the difference between the two curves is the generalization gap. [code: make_figures_ch05.py: fig_ch05_tradeoff]

Figure 40 shows this trade-off on a real experiment, where the complexity of the model is the degree of a polynomial fitted by least squares: the training error always decreases with the degree, whereas the test error first decreases (the model becomes flexible enough to capture the shape of the target), then increases (the model starts fitting the noise).

Remark (beyond the U-shaped curve) The U-shaped test error of Figures 39 and 40 is the classical picture, which is also known as the *bias-variance trade-off* Geman et al. (1992); Hastie et al. (2009). Very large models, such as modern neural networks, can however behave differently: when the number of parameters greatly exceeds the number of examples, the test error may decrease again after the interpolation threshold, a phenomenon called *double descent* Belkin et al. (2019). Such models generalize well although they can fit random labels perfectly Zhang et al. (2017), which shows that the number of parameters alone is not always a good measure of complexity; the stability and PAC-Bayesian approaches of this document are two ways of going beyond it.

Even if we can empirically assess the performance of a model on a test set, it is also important to provide *generalization guarantees*, *i.e.* to bound the true risk of the learned model using quantities observed at training time.

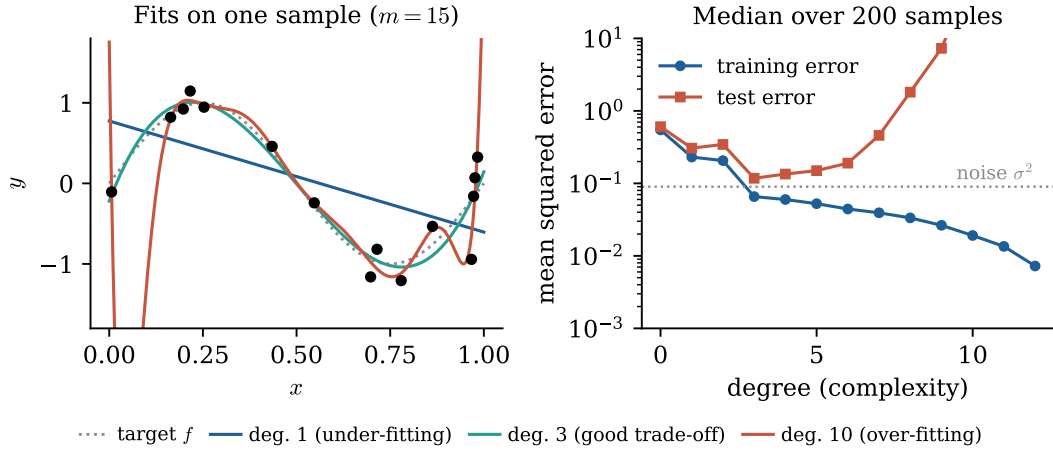


Figure 40: Under- and over-fitting with polynomial regression. The data are $y = \sin(2\pi x) + \epsilon$ with Gaussian noise of standard deviation 0.3 and $m = 15$ training points. Left: least-squares polynomials of degrees 1, 3 and 10 on one sample. Right: training and test mean squared errors as a function of the degree (log scale, median over 200 training samples). The test error is minimal for degrees around 3–5 and explodes when the number of parameters approaches m .
[`code: make_figures_ch05.py: fig_ch05_polyfit`]

The next subsection is dedicated to this point and presents generalization bounds based either on the complexity of the set of hypotheses or on some properties of the learning algorithm.

6.2 Generalization Bounds

A generalization bound controls the gap between the true risk and the empirical risk. It usually has the following general form, often referred to as a *Probably Approximately Correct* (PAC) bound [Valiant \(1984\)](#):

$$\mathbb{P}_{S \sim \mathcal{D}^m} \left(\left| \mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h) \right| \geq \varepsilon \right) \leq \delta, \quad (6.1)$$

where $\varepsilon \geq 0$ and $\delta \in [0, 1]$. Here, δ is an upper bound on the probability that the true risk deviates by at least ε from its empirical estimate: the estimate is *approximately* correct (up to ε) with *high probability* (at least $1 - \delta$). An immediate consequence is that, for a fixed sample size, the smaller ε , the greater (*i.e.* the closer to 1) the value of δ : we cannot be both very precise and very confident with a limited amount of data. Of course, the closer to 0 both δ and ε are, the more reliable our estimate is. Such bounds are usually derived using concentration inequalities, such as the Hoeffding [Hoeffding \(1963\)](#) or McDiarmid [McDiarmid \(1989\)](#) inequalities. Bound (6.1) can equivalently be rewritten as a probabilistic *bound of convergence* of the empirical estimate towards its mean [Vapnik and Chervonenkis \(1982\)](#); [Valiant \(1984\)](#):

$$\left| \mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h) \right| \leq \varepsilon(\delta, m),$$

where $1 - \delta$ is the confidence level of the bound and ε is a function of δ that decreases with the number m of training examples. Such a bound holds with probability

at least $1 - \delta$ over the draw of S , and the aim is to build a function $\varepsilon(\cdot)$ that converges to 0 as fast as possible.

In the following, we will see how such generalization bounds can be built, and that their convergence rate is often of order $\mathcal{O}(\sqrt{\ln(m)/m})$ or even $\mathcal{O}(1/\sqrt{m})$.

For a complete derivation of the presented bounds, the reader can refer to the references cited in the different parts and to the excellent book *Foundations of Machine Learning* Mohri et al. (2012, 2018); the book of Shalev-Shwartz and Ben-David Shalev-Shwartz and Ben-David (2014) is another very accessible introduction.

Historical note The PAC framework was introduced by Leslie Valiant in 1984 Valiant (1984), in the setting of computational learning theory, where one also asks the learning algorithm to run in polynomial time. The uniform convergence results it relies on are older: they were established by Vladimir Vapnik and Alexey Chervonenkis in the late 1960s and published in English in 1971 Vapnik and Chervonenkis (1971); Vapnik later developed them into a complete *statistical learning theory* Vapnik (1995, 1998), which also led to Support Vector Machines.

Uniform Deviation

As stated before, generalization bounds are based on the convergence of empirical quantities to their means Vapnik and Chervonenkis (1971) and rely on the law of large numbers (Theorem 1.6 in Section 1). Intuitively, the larger our training set, the closer the empirical risk of a fixed hypothesis h will be to its expected value, *i.e.*:

$$\lim_{m \rightarrow \infty} \frac{1}{m} \sum_{i=1}^m \ell(h(\mathbf{x}_i), y_i) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(h(\mathbf{x}), y)] \quad \text{almost surely.}$$

The law of large numbers only tells us that the empirical risk converges; Hoeffding's inequality (Proposition 1.14 in Section 1), recalled below, tells us *how fast*, and holds for every finite m . Figure 41 illustrates it for a fixed classifier.

However, the hypothesis h_S^* returned by the algorithm is *not* fixed in advance: it is chosen using the very same sample S . To handle this dependence, we look for bounds that are *uniform*, *i.e.* that hold simultaneously for all hypotheses $h \in \mathcal{H}$, hence in particular for h_S^* . Figure 42 shows why this is necessary: for each sample, the minimizer of the empirical risk is typically a hypothesis whose empirical risk happens to be lower than its true risk, so its training error is an optimistic estimate.

To get such a bound, we need to estimate the probability of the following event:

$$\left\{ \sup_{h \in \mathcal{H}} \left| \mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h) \right| \geq \varepsilon \right\}.$$

When $\mathcal{H}$ is finite and the loss takes its values in $[0, 1]$, the probability of this event can be bounded using Hoeffding's inequality, which states that for a fixed h , $\mathbb{P} \left(\left| \mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h) \right| \geq \varepsilon \right) \leq 2e^{-2m\varepsilon^2}$, together with the *union bound* (Proposition 1.8):

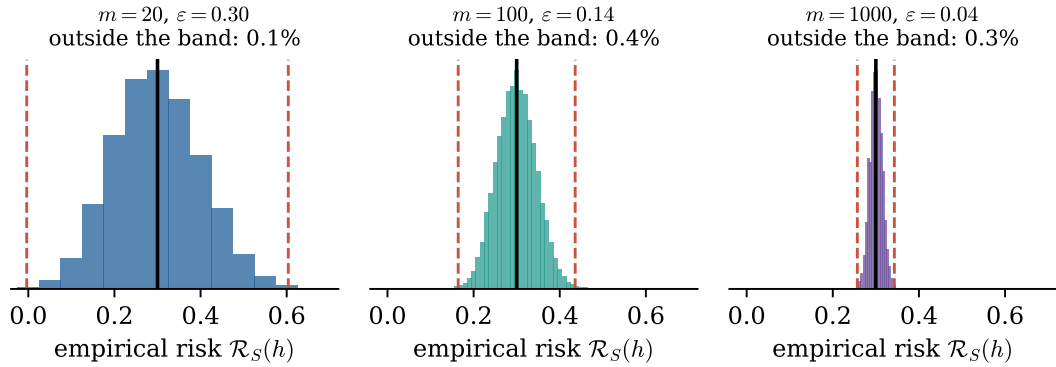


Figure 41: Concentration of the empirical risk of a *fixed* classifier h with true risk $\mathcal{R}^\ell(h) = 0.3$ (black line) and the 0-1 loss: histograms of $\mathcal{R}_S^\ell(h)$ over 20 000 samples S of size m . The dashed lines delimit the band $\mathcal{R}^\ell(h) \pm \varepsilon$ with $\varepsilon = \sqrt{\ln(2/\delta)/(2m)}$ and $\delta = 0.05$ given by Hoeffding's inequality: the empirical risk falls outside this band with frequency at most δ (in fact much less, Hoeffding's inequality being conservative), and the width of the band shrinks as $1/\sqrt{m}$. [code: `make_figures_ch05.py: fig_ch05_hoeffding`]

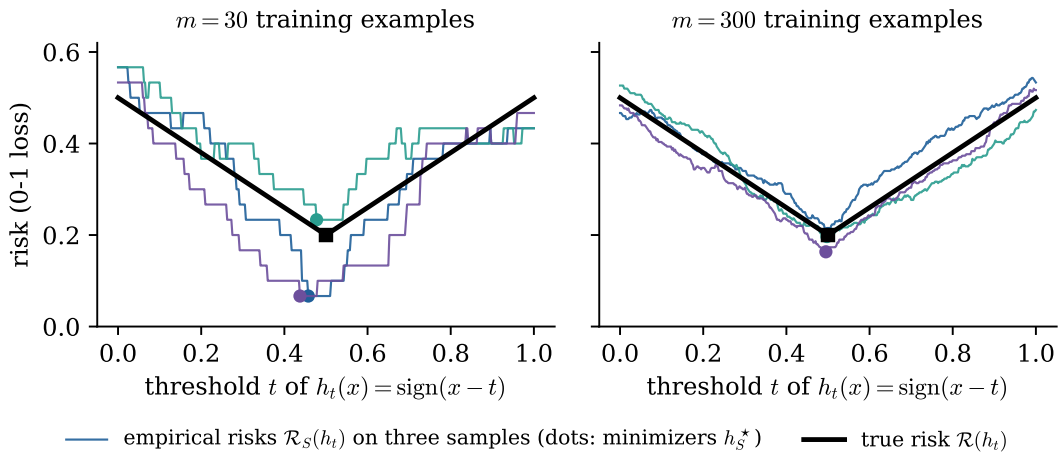


Figure 42: Empirical risk minimization over the threshold classifiers $h_t(x) = \text{sign}(x - t)$, for data with x uniform on $[0, 1]$ and $y = \text{sign}(x - 0.5)$ flipped with probability 0.2. Black: true risk (minimal at $t = 0.5$, square). Colored: empirical risks on three independent samples, with their minimizers h_S^* (dots). With $m = 30$, the empirical curves are far from the true one and their minimum underestimates the risk of h_S^* ; with $m = 300$, the whole curves are uniformly close to the true risk, so minimizing one amounts to almost minimizing the other. [code: `make_figures_ch05.py: fig_ch05_erm`]

$$\begin{aligned}
\mathbb{P}\left(\sup_{h \in \mathcal{H}} |\mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h)| \geq \varepsilon\right) &= \mathbb{P}\left(\bigcup_{h \in \mathcal{H}} \{|\mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h)| \geq \varepsilon\}\right) \\
&\downarrow \text{union bound: } \mathbb{P}(\cup_j A_j) \leq \sum_j \mathbb{P}(A_j) \\
&\leq \sum_{h \in \mathcal{H}} \mathbb{P}\left(|\mathcal{R}^\ell(h) - \mathcal{R}_S^\ell(h)| \geq \varepsilon\right) \\
&\downarrow \text{Hoeffding's inequality applied to each fixed } h \\
&\leq 2|\mathcal{H}| e^{-2m\varepsilon^2}.
\end{aligned}$$

Setting the right-hand side equal to δ and solving for ε leads to the following result [Bousquet et al. \(2004\)](#).

Theorem 6.1: Uniform Generalization Bound

Let $\mathcal{H}$ be a finite set of hypotheses, S a training sample of size m drawn i.i.d. from $\mathcal{D}$, ℓ a loss function taking its values in $[0, 1]$, and $\delta \in (0, 1]$. Then, with probability at least $1 - \delta$ over the draw of S , we have, for all $h \in \mathcal{H}$ simultaneously:

$$\mathcal{R}^\ell(h) \leq \mathcal{R}_S^\ell(h) + \sqrt{\frac{\ln |\mathcal{H}| + \ln(2/\delta)}{2m}}.$$

Example 6.2. Consider threshold classifiers $h_t(x) = \text{sign}(x - t)$ on the real line, where the threshold t is chosen among $|\mathcal{H}| = 1000$ values of a fixed grid, and the 0-1 loss. With $m = 10\,000$ training examples and $\delta = 0.05$, we get $\sqrt{(\ln 1000 + \ln 40)/20\,000} \approx \sqrt{(6.91 + 3.69)/20\,000} \approx 0.023$. Hence, with probability at least 95%, the true error rate of *any* of these classifiers, in particular the one minimizing the empirical error, exceeds its training error rate by at most 2.3 percentage points. Note that the size of $\mathcal{H}$ only enters through $\ln |\mathcal{H}|$: multiplying the number of hypotheses by 1000 only adds $\ln 1000 \approx 6.9$ in the numerator.

In a nutshell All the bounds of this section have the same structure:

$$\text{true risk} \leq \text{empirical risk} + \sqrt{\frac{\text{complexity} + \ln(1/\delta)}{m}} \quad (\text{up to constants}).$$

The first term is what we observe and minimize; the second one is the price to pay for having looked at only m examples. It grows with the richness of the class (or the instability of the algorithm) and with the required confidence $1 - \delta$, and it shrinks as $1/\sqrt{m}$: to divide it by 2, we need 4 times more data.

When the set $\mathcal{H}$ is infinite, *e.g.* the set of linear classifiers in $\mathbb{R}^d$, whose parameters belong to $\mathbb{R}^{d+1}$, the term $\ln |\mathcal{H}|$ is infinite and we need another way to measure the size, or complexity, of the set of hypotheses. A first way to measure this complexity is the VC-dimension, denoted by $VC(\mathcal{H})$ and introduced by Vapnik and Chervonenkis in the 1970s [Vapnik and Chervonenkis \(1971\)](#). It counts the largest number of points on which the hypotheses of $\mathcal{H}$ can realize *all* possible labelings.

Definition 6.3: VC-dimension

Let $\mathcal{H}$ be a set of binary classifiers with values in $\{-1, +1\}$. A set of points $\{\mathbf{x}_1, \dots, \mathbf{x}_n\} \subset \mathcal{X}$ is said to be *shattered* by $\mathcal{H}$ if, for every labeling $(y_1, \dots, y_n) \in \{-1, +1\}^n$, there exists $h \in \mathcal{H}$ such that $h(\mathbf{x}_i) = y_i$ for all $i = 1, \dots, n$. The VC-dimension $VC(\mathcal{H})$ is the size of the largest set of points that can be shattered by $\mathcal{H}$ (it is $+\infty$ if arbitrarily large sets can be shattered).

Example 6.3. Linear classifiers in $\mathbb{R}^2$ can shatter any set of 3 non-collinear points, but no set of 4 points (think of the “XOR” configuration, where the two diagonals of a square have opposite labels). Their VC-dimension is therefore 3. More generally, the VC-dimension of linear classifiers in $\mathbb{R}^d$ is $d + 1$, *i.e.* the number of parameters.

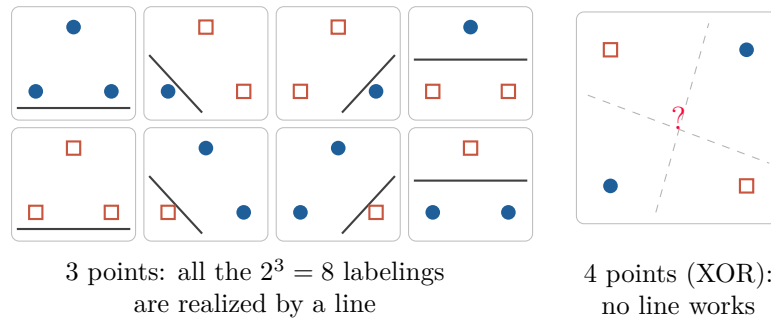


Figure 43: The VC-dimension of linear classifiers (half-planes) in $\mathbb{R}^2$ is 3. Left: three non-collinear points are *shattered*, *i.e.* each of the 8 possible labelings (blue disks: $+1$, red squares: -1) is realized by some line. Right: the XOR labeling of four points cannot be realized, since any line (dashed attempts) leaves points of both classes on one side; one can check that no set of four points in the plane can be shattered.

For the 0-1 loss, a set of hypotheses with finite VC-dimension enjoys a uniform generalization bound [Vapnik and Chervonenkis \(1971\)](#); [Mohri et al. \(2012\)](#): if $VC(\mathcal{H}) \leq m$, with probability at least $1 - \delta$, for all $h \in \mathcal{H}$,

$$\mathcal{R}^\ell(h) \leq \mathcal{R}_S^\ell(h) + \sqrt{\frac{2 VC(\mathcal{H}) \ln(e m / VC(\mathcal{H}))}{m}} + \sqrt{\frac{\ln(1/\delta)}{2m}}.$$

Compared to Theorem 6.1, the quantity $\ln |\mathcal{H}|$ is essentially replaced by $VC(\mathcal{H}) \ln m$, which gives a convergence rate in $\mathcal{O}(\sqrt{\ln(m)/m})$.

Remark Why does $\ln m$ appear? Even if $\mathcal{H}$ is infinite, on a given sample of m points its hypotheses can only produce a finite number of distinct labelings, and only these labelings matter for the empirical risk. Sauer’s lemma [Sauer \(1972\)](#) shows that this number is at most $(em/VC(\mathcal{H}))^{VC(\mathcal{H})}$ when $m \geq VC(\mathcal{H})$, *i.e.* it grows polynomially, and not exponentially, with m . Its logarithm, $VC(\mathcal{H}) \ln(em/VC(\mathcal{H}))$, plays the role of $\ln |\mathcal{H}|$ in the union bound.

Another way to measure the complexity of a set of hypotheses is the Rademacher complexity [Bartlett and Mendelson \(2003\)](#); [Koltchinskii and Panchenko \(2000\)](#), which is studied in more detail later in this document (see Definition 10.1). Informally, it measures the ability of the set of hypotheses to fit random noise: if the functions of

$\mathcal{H}$ can correlate well with random labels $\sigma_i \in \{-1, +1\}$, then $\mathcal{H}$ is rich, and a small empirical risk is less informative about the true risk.

However, these two measures are usually hard to compute, or even to estimate accurately, for most problems (except for simple ones, such as linear models), and the resulting bounds are often loose in practice. Fortunately, there exists another way to establish generalization bounds for convex regularized optimization problems: the *Uniform Stability* framework.

Uniform Stability

This framework is more recent [Bousquet and Elisseeff \(2002\)](#) and is not directly based on a measure of the complexity of the set of hypotheses. It relies instead on the concept of *stability* of the learning algorithm. Roughly speaking, an algorithm is *stable* if its output does not change significantly under a small modification of the training sample. The intuition is the following: if replacing one training example by another one barely changes the learned model, then the model cannot have “memorized” individual examples, and its training error should be a good proxy for its error on new data. More precisely, we focus on *uniform stability*: we look at the largest possible change in the loss, over all possible examples, when a single training example is replaced. A formal definition is given below.

Definition 6.4: Uniform Stability

A learning algorithm $\mathcal{A}$ has a uniform stability in $\frac{\beta}{m}$ with respect to a loss function ℓ , where β is a positive constant, if:

$$\forall S, \forall i, 1 \leq i \leq m, \sup_{\mathbf{z} \in \mathcal{X} \times \mathcal{Y}} |\ell(\boldsymbol{\theta}_S, \mathbf{z}) - \ell(\boldsymbol{\theta}_{S^i}, \mathbf{z})| \leq \frac{\beta}{m},$$

where S is a training sample of size m , $\boldsymbol{\theta}_S$ denotes the model parameters learned from S , and $\boldsymbol{\theta}_{S^i}$ the model parameters learned from the sample S^i obtained by replacing the i^{th} example $\mathbf{z}_i = (\mathbf{x}_i, y_i)$ of S by another example $\mathbf{z}'_i = (\mathbf{x}'_i, y'_i)$ drawn from $\mathcal{D}$, independently of S . Here, $\ell(\boldsymbol{\theta}_S, \mathbf{z})$ denotes the loss suffered at $\mathbf{z} = (\mathbf{x}, y)$ by the model learned from S^a .

^aNote that the authors also provide a definition of stability in which an example is removed instead of replaced. For the sake of convenience, we use the definition based on replacement throughout this document: indeed, it keeps the same number of training instances from one sample to the other.

The constant β depends on the properties of the loss function, but also on the regularization term of the minimization problem. The uniform stability property has been shown to hold for a wide range of minimization problems [Bousquet and Elisseeff \(2002\)](#). Using McDiarmid’s inequality (Proposition 1.17), we get the following generalization bound, whose rate of convergence is in $\mathcal{O}(1/\sqrt{m})$.

Theorem 6.2: Uniform Stability based Bound

Let $\delta \in (0, 1)$ and $m > 1$. For any algorithm with uniform stability β/m with respect to a loss function ℓ satisfying $0 \leq \ell \leq K$, with probability at least $1 - \delta$ over the random draw of S , we have:

$$\mathcal{R}^\ell(\boldsymbol{\theta}_S) \leq \mathcal{R}_S^\ell(\boldsymbol{\theta}_S) + \frac{2\beta}{m} + (4\beta + K)\sqrt{\frac{\ln(1/\delta)}{2m}},$$

where $\mathcal{R}^\ell(\cdot)$ is the true risk and $\mathcal{R}_S^\ell(\cdot)$ its empirical estimate over S .

The previous bound has a global convergence rate in $\mathcal{O}(1/\sqrt{m})$ and depends on two parameters: β , the uniform stability constant of the algorithm $\mathcal{A}$, and K , an upper bound on the loss function ℓ . Note that, unlike Theorem 6.1, it does not need to hold uniformly over $\mathcal{H}$: it only concerns the hypothesis actually returned by the algorithm.

The convexity of the problem is what makes it possible to compute β . The typical result is the following one.

Example 6.4. Consider a linear model $h_{\boldsymbol{\theta}}(\mathbf{x}) = \langle \boldsymbol{\theta}, \mathbf{x} \rangle$ with $\|\mathbf{x}\|_2 \leq \kappa$ for all $\mathbf{x} \in \mathcal{X}$, learned by solving

$$\boldsymbol{\theta}_S = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^m \ell(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) + \lambda \|\boldsymbol{\theta}\|_2^2,$$

where the loss ℓ is convex and σ -Lipschitz with respect to its first argument (*e.g.* the hinge loss, with $\sigma = 1$). Bousquet and Elisseeff [Bousquet and Elisseeff \(2002\)](#) showed that, when one example is removed, the loss changes by at most $\frac{\sigma^2 \kappa^2}{2\lambda m}$; replacing an example amounts to one removal followed by one addition, so this algorithm is uniformly stable with

$$\frac{\beta}{m} \leq \frac{\sigma^2 \kappa^2}{\lambda m}.$$

Plugging this value into Theorem 6.2 gives a bound in which everything is explicit: the smaller λ , the larger β , and the looser the bound.

This generalization bound is thus quite informative in practice, as both β and K can often be computed explicitly. Moreover, compared to the two previous frameworks, it is consistent with the practice of Machine Learning, since it takes into account the regularization term which is used in most minimization problems. As the previous example shows, the uniform stability constant β is a decreasing function of the hyper-parameter λ associated with the regularization term, whereas the empirical risk usually increases with λ . The impact of the hyper-parameter value is depicted in Figure 44, which shows that it is important to find the right value in order to obtain the best solution.

We will see that it is easy to build such bounds for convex regularized optimization problems, for both classification and regression tasks. It only requires some tools from linear algebra and the definition of a convex function.

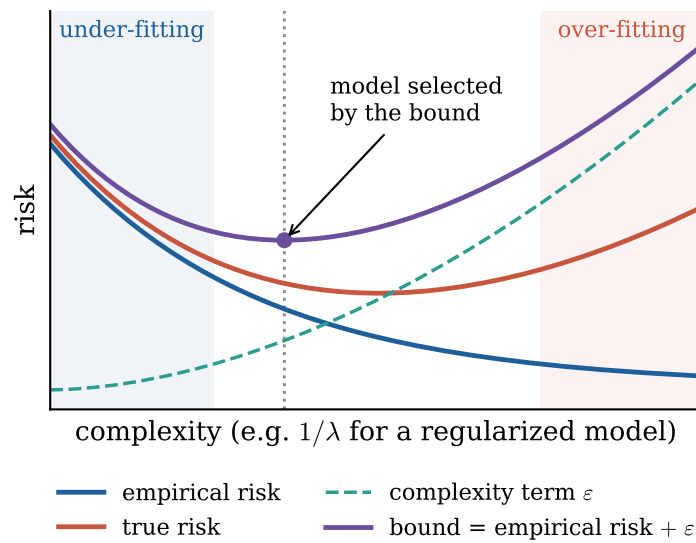


Figure 44: Evolution of the empirical and true risks and of a generalization bound according to the complexity of the model, *e.g.* the inverse $1/\lambda$ of the regularization hyper-parameter (schematic curves). The bound is the sum of the empirical risk and of a complexity term ε that increases with the complexity; it always lies above the true risk, and its minimizer gives a reasonable, if somewhat conservative, choice of the hyper-parameter. `[code: make_figures_ch05.py: fig_ch05_learnbound]`

To remember from Section 6

- The **true risk** $\mathcal{R}^\ell(h)$ is the expected loss of h on the unknown distribution $\mathcal{D}$; the **empirical risk** $\mathcal{R}_S^\ell(h)$ is its average on the sample S , an unbiased estimate for a *fixed* h .
- Minimizing the empirical risk over a too rich class leads to **over-fitting**; constraining the class too much leads to **under-fitting**. Structural and regularized risk minimization control the complexity through a penalty, tuned in practice by cross-validation.
- The excess risk of the learned model is the sum of an **approximation error** (due to $\mathcal{H}$) and an **estimation error** (due to the finite sample).
- A **generalization bound** states that, with probability at least $1 - \delta$, true risk $\leq$ empirical risk + $\varepsilon(\delta, m)$. Since h_S^* depends on S , the bound must hold uniformly over $\mathcal{H}$ (union bound, VC-dimension, Rademacher complexity) or exploit the **stability** of the algorithm.
- Typical rates are $\mathcal{O}(1/\sqrt{m})$ or $\mathcal{O}(\sqrt{\ln(m)}/m)$; for regularized convex problems, the stability constant behaves as $1/\lambda$.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

Empirical Risk Minimization

- Q6.1.** Why do we usually minimize a convex surrogate, such as the hinge loss, instead of the 0-1 loss?
- Q6.2.** A regressor predicts $h(\mathbf{x}_1) = 1$, $h(\mathbf{x}_2) = 2.5$ and $h(\mathbf{x}_3) = 0.5$ for the labels $y_1 = 1$, $y_2 = 2$ and $y_3 = 1$. Compute its empirical risk for the squared loss and for the absolute loss.
- Q6.3.** True or false? “Since $\mathcal{R}_S^\ell(h)$ is an unbiased estimator of $\mathcal{R}^\ell(h)$ for every h , the training error of h_S^* is an unbiased estimator of its true risk.”
- Q6.4.** How do the approximation error and the estimation error evolve when the class $\mathcal{H}$ grows? When the sample size m grows?
- Q6.5.** In Figure 40, why does the test error explode when the degree approaches $m = 15$?

Synthesis of the section

- Q6.6.** Why can the true risk not be computed in practice, and which quantity do we use instead?
- Q6.7.** Is the training error of h_S^* an unbiased estimate of its true risk?
- Q6.8.** In regularized risk minimization, what happens when $\lambda \rightarrow 0$? When $\lambda \rightarrow \infty$?
- Q6.9.** For a fixed classifier, $m = 1000$ examples and $\delta = 0.05$, what is the deviation ε given by Hoeffding’s inequality?
- Q6.10.** In Theorem 6.1, by how much must the sample size be multiplied to divide the deviation term by 2? By how much must $|\mathcal{H}|$ be multiplied to double $\ln |\mathcal{H}|$ when $|\mathcal{H}| = 1000$?
- Q6.11.** True or false? “For a fixed sample size, a larger δ gives a smaller ε .”
- Q6.12.** How does Theorem 6.1 change if the loss takes its values in $[0, K]$ instead of $[0, 1]$?
- Q6.13.** What is the VC-dimension of linear classifiers in $\mathbb{R}^5$?
- Q6.14.** True or false? “To prove that $VC(\mathcal{H}) \geq 3$, it suffices to exhibit one set of 3 points shattered by $\mathcal{H}$; to prove that $VC(\mathcal{H}) < 4$, it suffices to exhibit one set of 4 points that is not shattered.”
- Q6.15.** Which quantity does the uniform stability bound depend on, instead of the complexity of $\mathcal{H}$?
- Q6.16.** Why does the uniform stability bound not need to hold uniformly over $\mathcal{H}$?

7 Learning Procedure

Before presenting the different algorithms used in Supervised Learning, it is important to describe the way we learn a model. The aim of this section is to describe the right way to evaluate an algorithm and to tune its hyper-parameters, when the optimization problem involves one or several of them. We also present the classical performance measures used to evaluate a model in regression and in classification.

About this section

In this section you will learn to:

- *split a dataset into training, validation and test sets, and know which decision each of them is allowed to influence;*
- *tune hyper-parameters by (stratified) k-fold cross-validation, and read validation and learning curves to diagnose under- and over-fitting;*
- *recognize the most common forms of data leakage, which make an evaluation over-optimistic;*
- *compute and interpret the measures derived from a confusion matrix (accuracy, recall, precision, F_β , balanced accuracy, ...);*
- *build a ROC curve by moving a decision threshold, and know when a precision-recall curve is more informative.*

7.1 How to train and tune a Model

Imagine that we want to solve a given task in practice, such as a classification task, and that we have a labeled dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ for this purpose. Our goal is then to learn from our data a hypothesis h that solves this task as well as possible. We also want our model to generalize well to new data that it has not encountered yet. This last point is important because it is what allows us to deploy algorithms in production to solve tasks automatically.

The first question that arises is:

How can we measure the performance of our model on data that it has not encountered yet, starting from the data we have?

This is the first step of an experimental protocol in Machine Learning. In order to evaluate the performance of the model learned from our dataset, we start by splitting the dataset into two sets: a *training set* and a *test set* (see Figure 45).

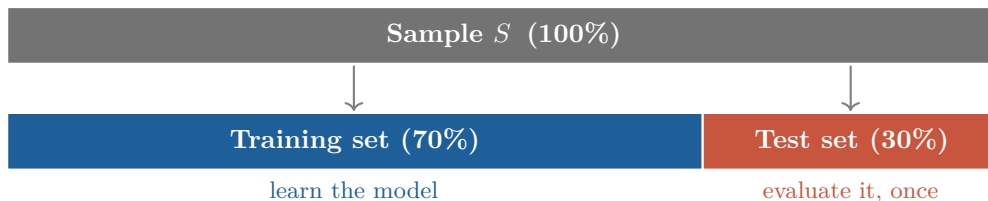


Figure 45: Illustration, with typical proportions, of how to split the sample into a training set and a test set.

Training set This set is used to learn the parameters of the model, *e.g.* the parameters of a linear separator or of a linear regression model. The algorithm has access to the features $\mathbf{x}_i$ **and** to the labels y_i of the data in order to be trained.

Test set This set is used to evaluate the model once it has been learned. The prediction is only based on the descriptors $\mathbf{x}_i$ of the data, and the labels are used a posteriori only, to check whether the predictions made by the model are correct or not, *i.e.* to evaluate the performance of the model on new data, that is, data that have not been seen during training.

Usually, we keep approximately 2/3 (or 70% to 80%) of the examples to train the model and the remaining examples to test it. It is important to keep a large amount of data to train the model, so that it is able to capture all the specificities of the data: indeed, the more data you have, the better they describe the underlying distribution. However, it is also important to keep enough examples in the test set so that the observed results are statistically significant. Note that the split should be random (and, in classification, it is common to preserve the class proportions in both sets, which is called a *stratified* split).

In a nutshell The test set plays the role of the final exam of a student. The training set contains the exercises studied during the semester. If the exam is made of exercises already solved in class, a good grade only proves that the student has a good memory, not that they have understood the course. Likewise, a model must be evaluated on examples that it has never seen, and the exam must remain sealed until the very end.

But what about hyper-parameter tuning?

The methodology presented above evaluates the model, but it does not solve the problem of hyper-parameter tuning: the test set must **never** be used to choose the hyper-parameters, otherwise the test performance would be an optimistic estimate of the generalization performance. The first thing we can do is to mimic the previous procedure, but on the *training set*! We thus obtain a sort of test set, called the *validation set*, which can be used to evaluate the performance of the model learned with each value of the hyper-parameter. More concretely, imagine that we aim to learn a hypothesis h which depends on a parameter θ and on a hyper-parameter λ . To tune the value of λ , we first select a range of candidate values for λ , denoted by λ_i , and the process works as follows:

1. Split the dataset into a training set and a test set.
2. Split the training set into a learning set and a validation set.
3. For each value λ_i :
 - (a) train your hypothesis (*i.e.* learn the parameter θ) with λ_i on the learning set;
 - (b) evaluate the performance of the learned model on the validation set. This gives you a performance p_i .
4. Keep the hyper-parameter value associated with the highest performance p_i ; we denote its index by $i_{\max}$.

5. Learn a model on *the whole training set* using the hyper-parameter value $\lambda_{i_{\max}}$.
6. Evaluate this model on the test set.

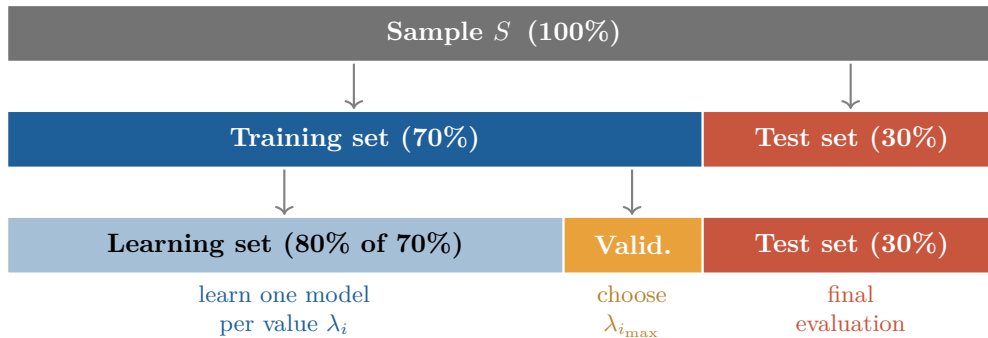


Figure 46: Illustration, with typical proportions, of how to split the sample into training and test sets, and then the training set into learning and validation sets.

The splitting process is illustrated in Figure 46. Note that, once the best hyper-parameter value is found, the model is trained again using a larger amount of data (the whole training set), in the hope of improving its performance. Figure 47 summarizes the role of each set: each of them is allowed to influence one, and only one, kind of decision.

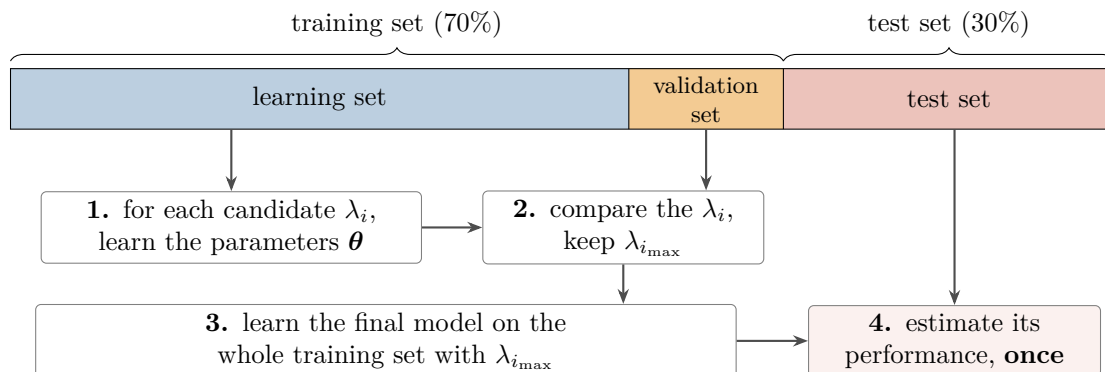


Figure 47: The hold-out protocol: which set is used for which decision. The parameters are learned on the learning set, the hyper-parameters are chosen on the validation set, and the test set is only used at the very end, to estimate the performance of the final model.

Pitfall: peeking at the test set If several models (or several values of a hyper-parameter) are evaluated on the test set and the best one is reported, the test set has in fact been used as a validation set, and the reported score is optimistic: among many noisy scores, the largest one is partly large by chance (this is made precise in Proposition 11.1 of Section 11). The same holds if the test set is used to choose the features, the preprocessing or the decision threshold of a classifier. Once a test score has been looked at, any decision taken on its basis turns the test set into a validation set.

Comparing the training and validation performances is also a way to diagnose the behavior of the model: if the model performs much better on the learning set than on

the validation set, it probably suffers from *over-fitting*; if its performance is poor on both sets, it probably suffers from *under-fitting*. This diagnosis is conveniently made with two plots, illustrated in Figure 48.

- The *validation curve* shows the training and validation scores as a function of a hyper-parameter that controls the complexity of the model (here, the maximal depth of a decision tree). The training score keeps increasing with the complexity, while the validation score first increases (the bias decreases) and then decreases (the variance increases): the best value lies at the top of the validation curve.
- The *learning curve* shows the same scores, for a fixed model, as a function of the size of the learning set. The gap between the two curves shrinks as more data become available; if the validation curve is still increasing at the right end of the plot, collecting more data is likely to improve the model.

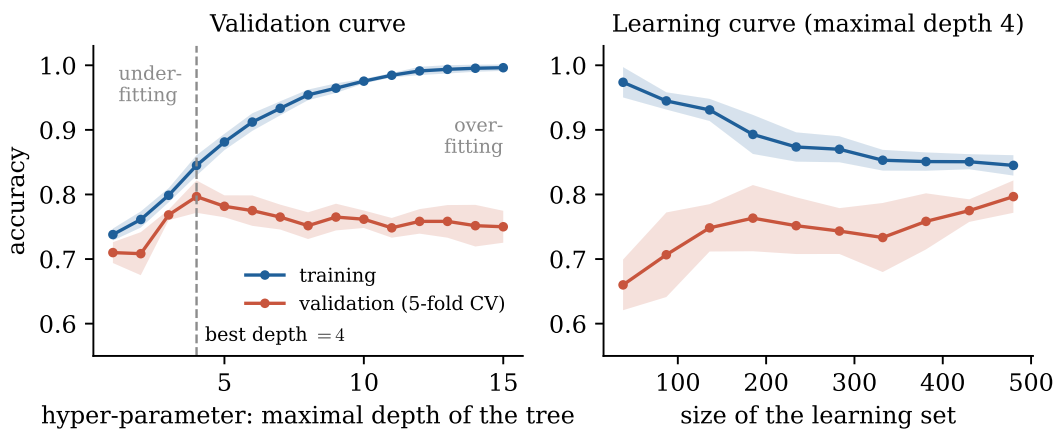


Figure 48: Left: validation curve of a decision tree on a synthetic dataset with 15% of noisy labels; the shaded areas represent $\pm$ one standard deviation over the 5 folds. Deep trees fit the training set almost perfectly but generalize worse. Right: learning curve of the selected tree; the gap between training and validation accuracy shrinks when the learning set grows. [code: `make_figures_ch06.py: fig_ch06_validation_curve`]

However, this tuning technique can be improved. Indeed, it has a flaw: the validation relies on a single, rather small, subset of the data, hence potentially on a small part of their distribution, and the result may depend a lot on this particular split.

In the previous section, we mentioned cross-validation as a way to tune hyper-parameters. **What does it consist of?** Let us illustrate it using Figure 49.

In this case, the training set is no longer split into two groups; instead, it is split into k groups, called folds. The tuning process is then the following:

1. For each value λ_i of the hyper-parameter:
 - (a) at the first round, select the first fold as the validation set and the $k - 1$ others as the learning set. This gives a first performance measure p_1 ;
 - (b) at the second round, select the second fold as the validation set and the $k - 1$ others as the learning set. This gives a second performance measure p_2 ;
 - (c) ...

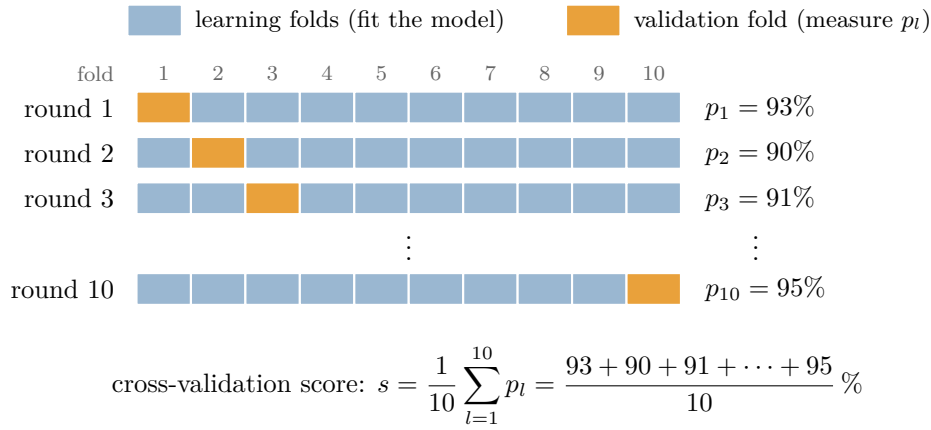


Figure 49: Illustration of the cross-validation process using a 10-fold cross-validation. The training set is divided into 10 folds, or groups. At each round, the model is learned on 9 folds (blue) and validated on the remaining one (orange), which gives a performance p_l , here an accuracy. The validation fold changes at each round, so that every example of the training set is used exactly once for validation. The cross-validation score is the average of the 10 values $p_1, \dots, p_{10}$.

- (d) at round k , select fold k as the validation set and the $k - 1$ others as the learning set. This gives a last performance measure p_k ;
 - (e) compute $s_i = \frac{1}{k} \sum_{l=1}^k p_l$, *i.e.* the average score over the k rounds.
2. Keep the hyper-parameter value associated with the highest score s_i ; we denote its index by $i_{\max}$.
 3. Learn a model on *the whole training set* using the hyper-parameter value $\lambda_{i_{\max}}$.
 4. Evaluate this model on the test set.

With this tuning method, every example of the training set is used once for validation, so that a larger part of the underlying distribution is explored. In practice, it therefore leads to a more reliable choice of the hyper-parameters. However, it requires learning as many models as there are folds (*i.e.* k) times the number of hyper-parameter values to test. Common choices are $k = 5$ or $k = 10$.

Remark on the number of folds The extreme case $k = m$, where each fold contains a single example, is called *leave-one-out* cross-validation. Since each model is learned on $m - 1$ examples, its estimate is almost unbiased, but it is costly and its variance can be large, because the m learned models are almost identical. Conversely, a small k learns models on noticeably fewer examples than the final one, which makes the estimate pessimistic. The values $k = 5$ or $k = 10$ are a good trade-off in practice [Kohavi \(1995\)](#); see [Arlot and Celisse \(2010\)](#) for a survey. In classification, the folds should be *stratified*, *i.e.* each of them should have the class proportions of the whole training set.

Example 7.1. Suppose that we want to choose the regularization constant λ of the lasso among the 5 values $\{10^{-3}, 10^{-2}, 10^{-1}, 1, 10\}$ using a 10-fold cross-validation. We then learn $5 \times 10 = 50$ models, compute 5 average scores $s_1, \dots, s_5$, keep the best value of λ ,

and learn a 51st model on the whole training set, which is finally evaluated once on the test set.

Remark on nested cross-validation When data are too scarce to set a test set aside, the cross-validation can itself be repeated inside another one: an *outer* cross-validation estimates the performance, and, in each of its rounds, an *inner* cross-validation chooses the hyper-parameters on the outer training folds only. This *nested cross-validation* Varma and Simon (2006) is detailed in Section 11.5 (see Figure 104).

Pitfall: data leakage through preprocessing Every step whose parameters are estimated from the data is part of the learning procedure, and must only see the learning data. A classical mistake is to standardize the features (with means and variances computed on the *whole* dataset), to impute the missing values, or to select the most relevant features *before* splitting the data or running the cross-validation: the validation and test examples then influence the model through the preprocessing. With standardization, the effect is often small; with feature selection, it can be dramatic Ambroise and McLachlan (2002), (Hastie et al., 2009, Section 7.10.2). In Figure 50, the labels are pure noise, so no classifier can do better than 50% of accuracy; yet, selecting the 20 features that are the most correlated with the labels on the whole dataset before running the cross-validation leads to an estimated accuracy above 90%. The cure is to fit the preprocessing *inside* each round of the cross-validation, e.g. with a pipeline (Section 11.4).

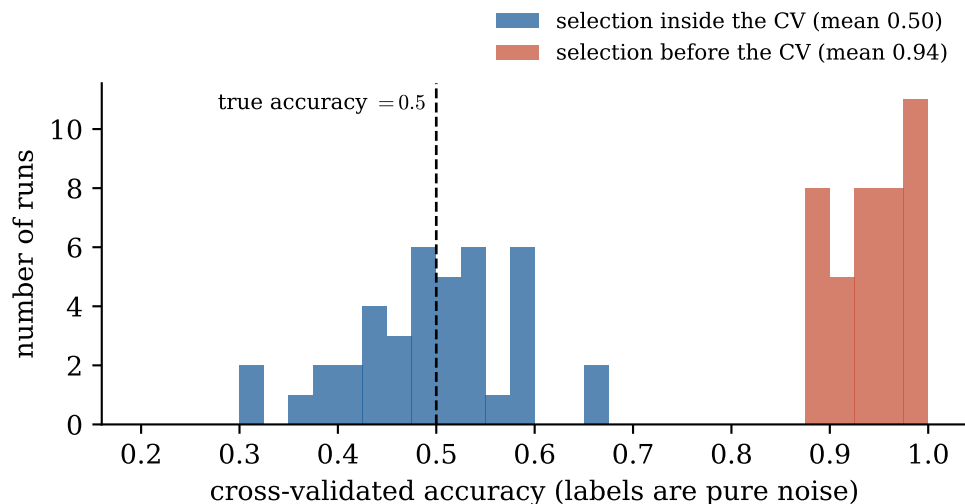


Figure 50: Feature selection must be part of the cross-validation. Each of the 40 runs draws $m = 50$ examples with 2000 Gaussian features and random labels, keeps the 20 features most correlated with the labels and estimates the accuracy of a logistic regression by 5-fold cross-validation. Red: the features are selected once, on the whole dataset (leakage). Blue: they are selected again on the learning folds of each round.[code: make_figures_ch06.py: fig_ch06_leakage]

So far, we have mentioned performance measures without ever explaining what they are. In Section 6.1, we already hinted at a potential performance measure when we talked about the number of errors made by a classifier (the error rate, whose complement

is called the accuracy). The next subsection is devoted to the presentation of classical performance measures in classification and regression.

7.2 Performance Measures

The choice of the performance measure depends on the nature of the problem, *i.e.* classification or regression, but also on what matters for the application at hand.

As in the previous section, we denote by $\mathbf{x}$ the feature vector and by y the label, or the real value to predict. We also denote by $\hat{y} = h(\mathbf{x})$ the output of a hypothesis h . All the measures below are computed on a set of m examples (typically a validation or a test set).

7.2.1 Regression tasks

In regression tasks, where the output space satisfies $\mathcal{Y} \subset \mathbb{R}$, we have already met several performance measures.

The most widely used one is the *Mean Squared Error* (MSE), which is the mean value, over all examples, of the squared difference between the predicted value $\hat{y}$ and the true value y :

$$MSE_h = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2.$$

Up to the constant $1/m$, the MSE is the squared ℓ_2 norm of the vector $\mathbf{y} - \hat{\mathbf{y}}$, *i.e.* $MSE_h = \frac{1}{m} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$. Its main advantage is that it is convex and differentiable, and can thus be directly optimized. One also often considers the *Root Mean Squared Error* (RMSE), which has the advantage of being expressed in the same unit as the target y , and is therefore easier to interpret:

$$RMSE_h = \sqrt{\frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2}.$$

The MSE is certainly the most used performance measure in regression. However, it has an important flaw: it is very sensitive to outliers. When an observation lies far from the regression line, its squared deviation, hence its contribution to the MSE, quickly becomes very large. When the MSE is used as the training criterion, this also introduces an important bias in the model, which is attracted towards the outliers (see Figure 51). That is why we often use another performance measure, the *Mean Absolute Error* (MAE), which is the mean value, over all examples, of the absolute difference between the predicted value $\hat{y}$ and the true value y :

$$MAE_h = \frac{1}{m} \sum_{i=1}^m |y_i - \hat{y}_i|.$$

Up to the constant $1/m$, the MAE is the ℓ_1 norm of the vector $\mathbf{y} - \hat{\mathbf{y}}$.

Example 7.2. Consider $m = 4$ examples with residuals $y_i - \hat{y}_i$ equal to 1, -1 , 1 and -1 . Then $MSE = MAE = 1$. If the last residual is replaced by -9 (an outlier), the MAE becomes $(1 + 1 + 1 + 9)/4 = 3$, whereas the MSE becomes $(1 + 1 + 1 + 81)/4 = 21$: the single outlier accounts for more than 95% of the MSE.

Other measures of the *quality* of a regression model were met in the Linear Models course, such as the AIC and BIC criteria, or the coefficient of determination R^2 , defined by

$$R^2 = 1 - \frac{\sum_{i=1}^m (y_i - \hat{y}_i)^2}{\sum_{i=1}^m (y_i - \bar{y})^2}, \quad \text{where } \bar{y} = \frac{1}{m} \sum_{i=1}^m y_i,$$

which compares the error of the model to that of the constant predictor $\bar{y}$: the closer R^2 is to 1, the better the model.

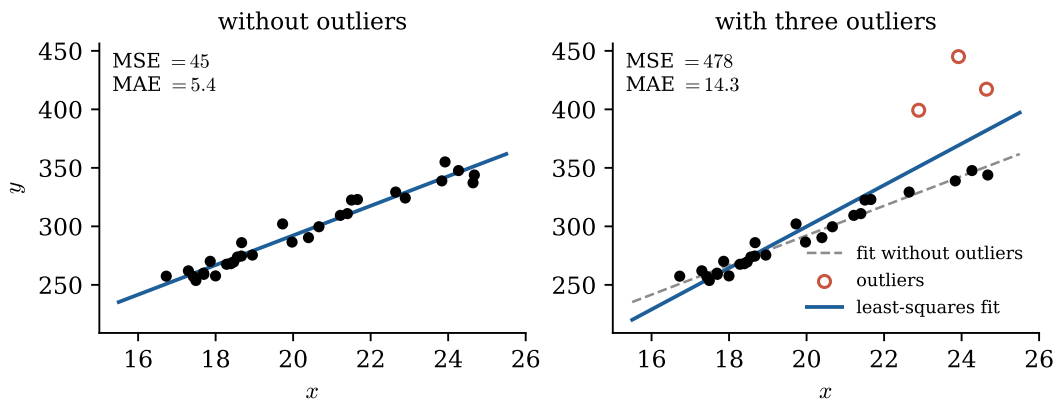


Figure 51: Sensitivity of the MSE to outliers. Left: a least-squares regression line fitted on 30 points without outliers. Right: the same data in which three responses have been shifted upwards (open circles); the least-squares line (solid) is attracted towards them and moves away from the line fitted without outliers (dashed). The MSE of the fitted line is multiplied by more than 10 (from 45 to 478), the MAE only by about 2.6 (from 5.4 to 14.3). [code:

make_figures_ch06.py: fig_ch06_mse_outliers]

7.2.2 Classification tasks

In this section, we present different metrics, or performance measures, that are suited to the binary setting. Some of them can easily be extended to the multi-class setting, as explained at the end of this section.

We consider a binary classification problem with two classes: $y = -1$ and $y = 1$. The class $y = 1$ is called the positive class and the class $y = -1$ the negative one⁸. In many applications (fraud, disease, spam), the positive class is the class of interest and is also the minority class. In this setting, we can compute several performance measures from a *Confusion Matrix*, an example of which is given in Table 2. An example from the positive class that is predicted positive by the learned model is called a *True Positive (TP)*; if it is misclassified, it is called a *False Negative (FN)*. Similarly, an example from the negative class that is well classified is called a *True Negative (TN)*, and a *False*

⁸We will sometimes use the label 0 to denote the negative class, for the sake of simplicity, when presenting some of the algorithms.

Positive (FP) otherwise. With a slight abuse of notation, TP, FN, FP and TN also denote the *numbers* of such examples, so that $TP + FN + FP + TN = m$.

	Predicted positive ($h(\mathbf{x}) = 1$)	Predicted negative ($h(\mathbf{x}) = 0$)
Actual positive ($y = 1$)	True Positive (TP)	False Negative (FN)
Actual negative ($y = 0$)	False Positive (FP)	True Negative (TN)

Table 2: Confusion matrix for a binary classification task. The label is denoted by y and the prediction by $h(\mathbf{x})$.

Standard Measures The most common performance measure used to evaluate an algorithm is the *Accuracy*, which is the complement of the *Error rate*. Using the notations of Table 2, they are defined as follows.

Definition 7.1: Accuracy - Error rate

Let us consider a set of m examples and a hypothesis h which leads to a given confusion matrix on these examples. The Accuracy of h is then defined by:

$$\text{Accuracy} = \frac{TP + TN}{m},$$

and the Error rate is defined by:

$$\text{Error rate} = 1 - \text{Accuracy} = \frac{FP + FN}{m}.$$

Although this measure is the most used one to evaluate the performance of algorithms, it is not relevant in all settings, *e.g.* when the dataset is *imbalanced*, *i.e.* when one class is much less represented than the other.

Example 7.3. Let us consider a dataset with $m = 100,000$ examples. Suppose that the number of positive examples is $m_+ = 1,000$ and the number of negative examples is $m_- = 99,000$. Then a trivial classifier, which predicts all the examples as negative, achieves an accuracy of 99%. This looks good at first glance, but this solution is not satisfactory because the learned hypothesis does not detect a single example of the positive class.

To overcome the issue raised in the above example, we need another performance measure which takes into account the ability of the model h to capture the minority class.

Definition 7.2: Sensitivity or Recall

Let us consider a set of m examples and a hypothesis h which leads to a given confusion matrix on these examples. The *Sensitivity*, also known as *Recall* or *True*

Positive Rate, of a classifier h is defined as:

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

By definition, the Recall is the proportion of examples of the class of interest that the model is able to retrieve. The higher this value (*i.e.* the closer to 1), the more examples of the minority class are retrieved (the trivial classifier of the previous example has a Recall equal to 0). Because it measures the ability of a model to detect the examples of the minority class, the Recall is a good candidate for a performance measure in imbalanced scenarios, *e.g.* in medical diagnosis, where missing a patient with a serious disease can be fatal.

In other applications, such as in banking, it is important to be confident in positive predictions: *e.g.* when an individual is denied a more or less fundamental right (such as a loan), the decision had better be the right one, otherwise there could be serious consequences for the company or organization that made the decision. In this case, we want the model to be reliable when it predicts the positive class, *i.e.* we focus on the Precision of the model.

Definition 7.3: Precision

Let us consider a set of m examples and a hypothesis h which leads to a given confusion matrix on these examples. The *Precision*, also called *Positive Predictive Value*, of a hypothesis h is defined as:

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

The Precision is the proportion of correct predictions among the examples predicted positive. Note that Recall and Precision are usually in tension: predicting more examples as positive tends to increase the Recall and to decrease the Precision.

In a nutshell Both measures share the same numerator, TP , but they look at the confusion matrix of Table 2 in two different directions. The Recall reads the first **row**: among the examples that *are* positive, how many did we find? The Precision reads the first **column**: among the examples that we *predicted* positive, how many are really positive? A search engine returning a single, relevant page has a perfect Precision but a poor Recall; one returning the whole web has a perfect Recall but a poor Precision.

Other performance measures Konukoglu and Ganz (2014); Branco et al. (2016) based on the confusion matrix of h are given below:

$$\begin{aligned} \text{True Negative Rate} &= \frac{TN}{TN + FP}, & \text{False Negative Rate} &= \frac{FN}{FN + TP}, \\ \text{False Positive Rate} &= \frac{FP}{FP + TN}, & \text{Negative Predictive Value} &= \frac{TN}{TN + FN}. \end{aligned}$$

Such measures can be seen as complementary to the Precision and the Recall. Note that the True Negative Rate is also known as *Specificity*, and that False Positive Rate = $1 - \text{Specificity}$ and False Negative Rate = $1 - \text{Recall}$. As shown before, the choice of the metric depends on the user's preferences [Torgo and Ribeiro \(2007\)](#); [Torgo and Lopes \(2011\)](#).

In imbalanced settings, such as fraud detection, it is important to take into account both the Precision and the Recall of the model. This is the goal of the F-measure, introduced in the seventies [Rijsbergen \(1979\)](#).

A Versatile Measure: the F-measure The F-measure is a good performance measure when a user wants to focus on the behavior of a model on a minority class. It is widely used in information retrieval [Sanderson \(1994\)](#) and, of course, in fraud and anomaly detection [Gee \(2014\)](#); [Bahnsen et al. \(2014\)](#); [Bolton and Hand \(2002\)](#). It is defined as a weighted harmonic mean of the Precision and the Recall, and depends on a parameter $\beta > 0$. Denoting by $P = TP + FN$ the number of positive examples, it reads:

$$\begin{aligned} F_\beta &= \frac{(1 + \beta^2) \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}} \\ &= \frac{(1 + \beta^2) TP}{(1 + \beta^2) TP + \beta^2 FN + FP} = \frac{(1 + \beta^2)(P - FN)}{(1 + \beta^2)P - FN + FP}. \end{aligned}$$

An illustration of the F-measure is given in Figure 52. The F-measure is a flexible measure: by modifying the value of β , the user controls the relative importance of the Precision and the Recall. If $\beta = 1$, the same weight is given to both. If the user wants to give more importance to the Recall, they can choose a value of β greater than one; by choosing $\beta < 1$, they give more importance to the Precision.

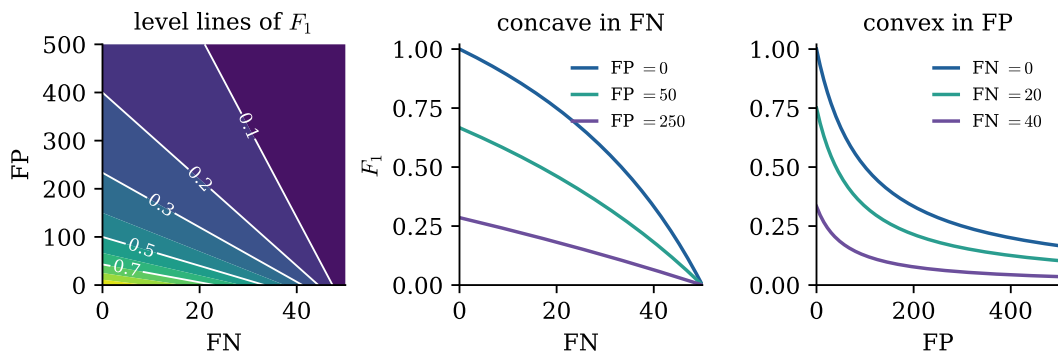


Figure 52: The F-measure F_1 as a function of the numbers of false negatives (FN) and false positives (FP), with 50 positive and 500 negative examples. Left: map of F_1 with some of its level lines, which are straight lines. Middle: sections at a fixed FP, which are concave functions of FN. Right: sections at a fixed FN, which are convex functions of FP. The F-measure is therefore neither convex nor concave in the entries of the confusion matrix. [code: `make_figures_ch06.py: fig_ch06_fmeasure`]

Example 7.4. Let us consider a sample of 1000 instances in which the minority class represents 1% of the data, so that we have 10 positive examples for 990 negative ones. Let us also consider two hypotheses, h_1 and h_2 , with the following confusion matrices:

h_1		Predicted positive	Predicted negative
	Actual positive	TP = 3	FN = 7
	Actual negative	FP = 0	TN = 990
h_2		Predicted positive	Predicted negative
	Actual positive	TP = 9	FN = 1
	Actual negative	FP = 6	TN = 984

Both classifiers have the same error rate, $7/1000 = 0.7\%$, which is less than 1%. The first classifier, h_1 , has a Precision equal to 1 and a Recall equal to 0.3, while for the second classifier, h_2 , these two values are respectively equal to $9/15 = 0.6$ and 0.9. These two classifiers lead to very different F-measures (with $\beta = 1$): $2 \times 0.3/1.3 \approx 0.46$ for the first hypothesis and $2 \times 0.54/1.5 = 0.72$ for the second one. Despite the perfect Precision reached by the first classifier, its F-measure remains lower than the one achieved by h_2 .

Despite its usefulness in imbalanced scenarios, the F-measure remains, compared to the Accuracy, hard to optimize. It presents two main drawbacks: (i) it is non-linear in the entries of the confusion matrix and (ii) it is neither convex nor concave in these entries, as depicted in Figure 52: for a fixed FP, it is a concave function of FN, while for a fixed FN, it is a convex function of FP. Because of (i), the F-measure cannot be written as an average of losses over the examples, which makes it hard to derive generalization guarantees for it. Moreover, like the accuracy, it is computed from counts of errors and is thus piecewise constant with respect to the parameters of the model, so standard gradient descent algorithms cannot be used to optimize it directly. Because of (ii), the optimization problems obtained when the F-measure is expressed as a function of the parameters of the model are non-convex, and an optimization algorithm can get stuck in local optima that may be far from the optimal solution. However, the literature offers many studies and algorithms dealing with this task [Zhao et al. \(2013\)](#); [Busa-Fekete et al. \(2015\)](#).

7.2.3 Other Performance Measures

The *Class Weighted Accuracy*, denoted by *CWA* and proposed by [Cohen et al. \(2006\)](#), is similar in spirit to the F-measure. However, it no longer involves the Precision but the Specificity instead, which, like the Precision, penalizes false positives. It is expressed as a convex combination of the Sensitivity (*i.e.* the Recall) and the Specificity: for any $\alpha \in [0, 1]$,

$$CWA = \alpha \times \text{Sensitivity} + (1 - \alpha) \times \text{Specificity}.$$

As for the parameter β of the F-measure, the user chooses the value of the parameter α . With $\alpha = 1/2$, it is called the *balanced accuracy*; the trivial classifier of the example above then gets a score of only 0.5.

Another evaluation metric used in imbalanced scenarios is the G-mean. As for the Class Weighted Accuracy, its expression depends on both Sensitivity and Specificity: it is the square root of the product (*i.e.* the geometric mean) of these two quantities:

$$\text{G-mean} = \sqrt{\text{Specificity} \times \text{Sensitivity}}.$$

This measure is used in imbalanced scenarios because it takes into account the performance on both classes, while giving the same importance to each class.

A last popular evaluation metric used in imbalanced scenarios is the AUC ROC Metz (1978); Cortes and Mohri (2004), the “Area Under the Receiver Operating Characteristic curve”. This metric is used when the learned model returns a score, such as a confidence in its prediction, and an example is predicted positive when its score exceeds a threshold. Each value of the threshold gives a pair (False Positive Rate, Recall); the ROC curve is obtained by plotting the Recall (True Positive Rate) against the False Positive Rate for all possible thresholds, *i.e.* for all the scores obtained by the model on the examples (see Figure 54). The AUC is the area under this curve: it equals 1 for a perfect ranking and 0.5 for a random one, and it can be interpreted as the probability that a randomly drawn positive example gets a higher score than a randomly drawn negative example. The closer to 1 the AUC, the better the model.

More precisely, let $f(\mathbf{x})$ denote the score of the model, and let $h_t(\mathbf{x}) = 1$ if $f(\mathbf{x}) \geq t$ and -1 otherwise be the classifier obtained with the threshold t . Denoting by m_+ and m_- the numbers of positive and negative examples, the ROC curve is the set of points

$$(\text{FPR}(t), \text{TPR}(t)) = \left(\frac{1}{m_-} \sum_{i: y_i = -1} \mathbb{1}[f(\mathbf{x}_i) \geq t], \frac{1}{m_+} \sum_{i: y_i = 1} \mathbb{1}[f(\mathbf{x}_i) \geq t] \right), \quad t \in \mathbb{R}.$$

When t decreases from $+\infty$ to $-\infty$, more and more examples are predicted positive, and the point moves from $(0, 0)$ (everything is predicted negative) to $(1, 1)$ (everything is predicted positive), as illustrated in Figure 53. Each positive example that passes the threshold moves the curve up by $1/m_+$, and each negative example moves it right by $1/m_-$: the curve is therefore a staircase, and it only depends on the *ranking* of the examples induced by the scores, not on the values of the scores themselves. Counting, for each positive example, the proportion of negative examples with a lower score, one obtains the equivalent expression

$$\text{AUC} = \frac{1}{m_+ m_-} \sum_{i: y_i = 1} \sum_{j: y_j = -1} \mathbb{1}[f(\mathbf{x}_i) > f(\mathbf{x}_j)]$$

(ties being counted for $1/2$), which is exactly the probabilistic interpretation given above Fawcett (2006).

Historical note ROC curves come from *signal detection theory*, developed during the Second World War to analyze radar signals: an operator had to decide whether a blip on the screen was an enemy aircraft (a “signal”) or noise, and the curve described the trade-off between detections and false alarms, hence the name *Receiver Operating Characteristic*. They were then adopted in psychology and in medical diagnosis Metz (1978), before becoming a standard tool in machine learning Fawcett (2006).

Compared to the previous measures, the ROC curve allows us to visualize the performance of the model for all thresholds at once. Furthermore, it is often used to choose the appropriate threshold for the task at hand. It also makes it possible to plot several models on a single graph and to choose the one best suited to a given purpose. The ideal model is the one that achieves a True Positive Rate equal to 1 while the False

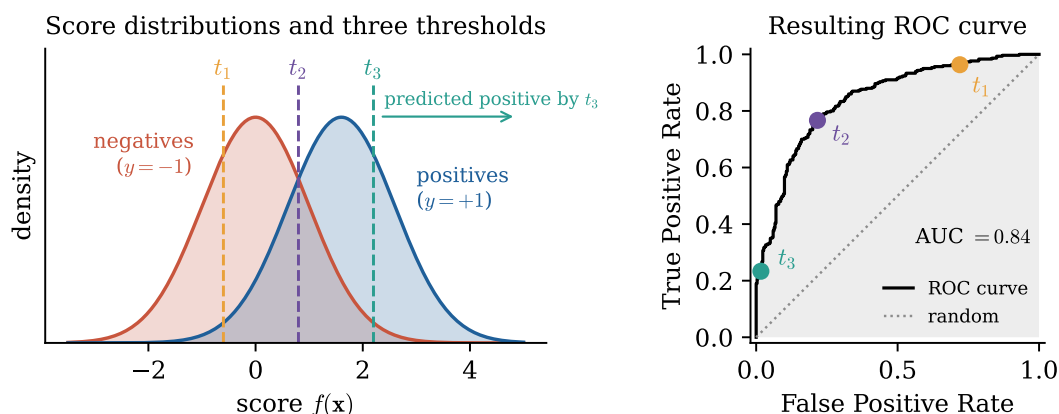


Figure 53: Construction of a ROC curve. Left: distribution of the scores of the negative and positive examples. A threshold t predicts positive every example on its right; the proportion of positive (resp. negative) examples on its right is the True (resp. False) Positive Rate. Right: each threshold gives one point of the ROC curve; a high threshold t_3 gives few false positives but a low recall, a low threshold t_1 the opposite. The more the two distributions overlap, the closer the curve to the diagonal. [code: make_figures_ch06.py: fig_ch06_roc_construction]

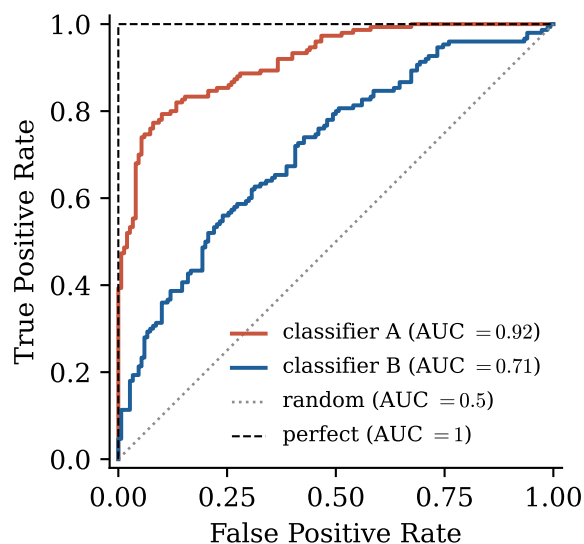


Figure 54: ROC curves and AUC of two classifiers evaluated on the same 300 examples. The curve of classifier A (red) lies above that of classifier B (blue) almost everywhere, hence a larger AUC. The dotted diagonal corresponds to a random ranking, the dashed line to a perfect one. [code: make_figures_ch06.py: fig_ch06_roc_two_models]

Positive Rate remains equal to 0, *i.e.* a model for which all the examples of the minority class have a greater score than the examples of the majority class.

There exist other measures used in imbalanced scenarios, such as the Average Precision, the H-measure or the Precision-Recall curve, to cite a few [Frery et al. \(2017\)](#); [Ferri et al. \(2009\)](#); [Jeni et al. \(2013\)](#); [Branco et al. \(2016\)](#); [Behl et al. \(2014\)](#). Although most of them were designed for imbalanced tasks, they can also be used to evaluate performance in standard contexts. The *Precision-Recall curve* is built exactly like the ROC curve, by moving the threshold t , but it plots the Precision against the Recall; the area under this curve is usually estimated by the *Average Precision* (AP).

Pitfall: a ROC curve can hide a poor Precision The True and False Positive Rates are each computed within one class, so the ROC curve and the AUC do not depend on the class proportions. This is a strength when comparing models across datasets, but it can be misleading in a strongly imbalanced setting: a False Positive Rate of 5% looks small, yet with 99 negatives for each positive it means about five false alarms for each positive example of the dataset. The Precision, on the contrary, involves both classes and directly reveals this problem. In Figure 55, the scores of each class have the same distribution in both datasets: the ROC curves are identical, whereas the Precision-Recall curves are very different. When the positive class is rare, the Precision-Recall curve is therefore more informative [Davis and Goadrich \(2006\)](#); [Saito and Rehmsmeier \(2015\)](#). Note also that the curve of a random classifier is the diagonal in ROC space, but a horizontal line at the proportion of positives in Precision-Recall space.

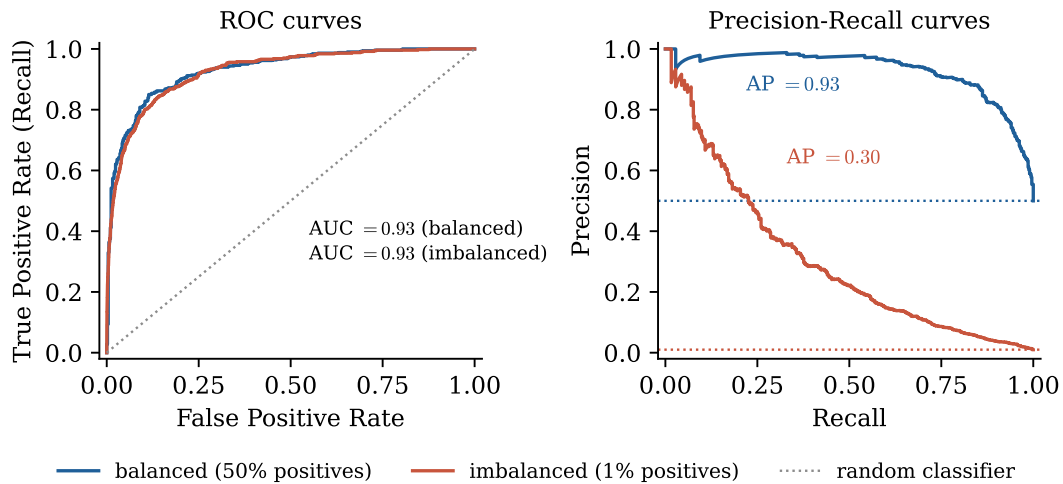


Figure 55: ROC curves (left) and Precision-Recall curves (right) of the same scoring function on a balanced dataset (500 positives, 500 negatives) and on an imbalanced one (500 positives, 49,500 negatives). The AUC is unchanged, but the Average Precision drops from 0.93 to 0.30. The dotted horizontal lines give the Precision of a random classifier, *i.e.* the proportion of positives.

[code: make_figures_ch06.py: fig_ch06_roc_vs_pr]

Multi-class setting. The Accuracy and the Error rate extend directly to $q > 2$ classes, as the proportions of well-classified and misclassified examples, and the confusion matrix becomes a $q \times q$ table. The other measures (Recall, Precision, F-measure, ...)

are computed for each class c by considering c as the positive class and all the other classes as the negative one (*one-versus-rest*); the q values obtained are then averaged, either with equal weights (*macro-averaging*) or by first summing the TP, FP and FN counts over the classes (*micro-averaging*).

Now that we have explained how to learn, tune and evaluate a model in general, it is time to present some fundamental algorithms of supervised Machine Learning.

To remember from Section 7

- The parameters are learned on the learning set, the hyper-parameters are chosen on a validation set (or by k -fold cross-validation on the training set), and the test set is used **once**, at the end, to estimate the performance of the final model.
- Validation and learning curves compare the training and validation scores to diagnose under-fitting (both are poor) and over-fitting (large gap).
- Any step estimated from the data (scaling, imputation, feature selection, resampling) belongs to the learning procedure and must be fitted on the learning data only; otherwise, there is a data leakage and the evaluation is optimistic.
- In regression, the MSE penalizes large errors much more than the MAE, which is more robust to outliers.
- In classification, the accuracy is misleading on imbalanced data; the Recall (row of the confusion matrix), the Precision (column), the F_β -measure, the balanced accuracy or the G-mean focus on the minority class.
- The ROC curve is obtained by moving the threshold of a scoring function; its area is the probability that a positive example is ranked above a negative one. When positives are rare, the Precision-Recall curve is more informative.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

How to train and tune a Model

- Q7.1.** Why must the hyper-parameters never be chosen on the test set?
- Q7.2.** After a 10-fold cross-validation has selected $\lambda_{i_{\max}}$, which model is evaluated on the test set?
- Q7.3.** A model reaches 99% of accuracy on the learning set and 70% on the validation set. What is your diagnosis, and what could you try?
- Q7.4.** What is wrong with standardizing the features of the whole dataset and then running a cross-validation?
- Q7.5.** Two hyper-parameters are tuned on a grid of 4×3 values by 5-fold cross-validation. How many models are learned, including the final one?
- Q7.6.** True or false? Leave-one-out cross-validation is always the best choice, since each model is learned on $m - 1$ examples.

- Q7.7.** On a learning curve, the training and validation accuracies are both around 65% and are almost flat for the largest learning sets. Will collecting more data help?
- Q7.8.** Why is the best average score obtained during a cross-validated grid search not a reliable estimate of the performance of the selected model?

Synthesis of the section

- Q7.9.** On a dataset with 1% of positives, what are the accuracy, the Recall and the balanced accuracy of the classifier that always predicts the negative class?
- Q7.10.** A model has an AUC of 0.5. What does it mean? Why can a model with an AUC of 0.95 still be useless when positives are very rare?
- Q7.11.** The residuals of a regression model on four examples are 2, -2, 0, 0. Compute its MSE, RMSE and MAE.
- Q7.12.** True or false? The coefficient R^2 of a model computed on a test set always lies in $[0, 1]$.
- Q7.13.** A classifier gives $TP = 40$, $FN = 10$, $FP = 20$ and $TN = 930$. Compute its accuracy, Recall, Precision and F_1 .
- Q7.14.** True or false? The F_1 -measure always lies between the Precision and the Recall.
- Q7.15.** What does F_β become when $\beta \rightarrow 0$ and when $\beta \rightarrow +\infty$?
- Q7.16.** What happens to the ROC curve and to the AUC if all the scores $f(\mathbf{x}_i)$ are replaced by $\exp(f(\mathbf{x}_i))$? By $-f(\mathbf{x}_i)$?

8 Supervised Algorithms

About this section

Guiding question. *Given a labeled sample, which classical algorithms can we use, what do they optimize, and how do their hyperparameters control their behavior?*

In this section you will learn to:

- *explain why convex surrogate losses replace the 0-1 loss, and recognize the hinge, logistic and exponential losses;*
- *use the k -NN rule, and anticipate the effect of k , of the scaling of the features and of the dimension;*
- *derive the hard and soft margin SVM problems and their dual, and use the kernel trick to obtain nonlinear classifiers;*
- *fit linear, logistic and multinomial logistic regression models, with L_1 or L_2 regularization;*
- *build a kernel regression estimator and a decision tree, and relate their hyperparameters (bandwidth, depth) to the bias-variance trade-off.*

8.1 Surrogate losses

In Section 6, generalization bounds based on uniform deviations were presented for a loss function ℓ taking its values in $[0, 1]$, typically the 0-1 loss in the context of classification. However, as we have already pointed out, this loss is hard to optimize from an algorithmic point of view because it is neither convex nor differentiable. We therefore rather use convex (and sometimes differentiable) relaxations of it, called *surrogate losses*, to learn our hypothesis $h(\cdot, \cdot)$, whose two arguments are respectively the parameters $\mathbf{w}$ of the model and an input $\mathbf{x}$. When the parameters are clear from the context, we simply write $h(\mathbf{x})$. The 0-1 loss and some of its surrogates are represented in Figure 56; they are all written as functions of the quantity $yh(\mathbf{x})$, which is positive when the prediction is correct and negative otherwise.

Among these surrogates, the *hinge loss* is the one used to train a Support Vector Machine Boser et al. (1992); Vapnik and Cortes (1995). It is used for classification tasks and is defined by

$$\begin{aligned}\ell(h(\mathbf{w}, \mathbf{x}), y) &= [1 - yh(\mathbf{w}, \mathbf{x})]_+ \\ &= \max(0, 1 - yh(\mathbf{w}, \mathbf{x})).\end{aligned}$$

Another well-known surrogate is the *exponential loss*, which is widely used in the context of boosting Friedman et al. (2000):

$$\ell(h(\mathbf{w}, \mathbf{x}), y) = \exp(-yh(\mathbf{w}, \mathbf{x})).$$

Compared to the hinge loss, the exponential loss penalizes errors much more heavily (the penalty grows exponentially with the size of the error) and it never equals 0.

Finally, the *logistic loss*, used to train a logistic regression model Mohri et al. (2012), is defined by

$$\ell(h(\mathbf{w}, \mathbf{x}), y) = \frac{1}{\ln(2)} \ln(1 + \exp(-yh(\mathbf{w}, \mathbf{x}))).$$

The constant $1/\ln(2)$ is only a normalization ensuring that the loss equals 1 when $yh(\mathbf{w}, \mathbf{x}) = 0$, as the 0-1 loss does; it is often omitted. Note that this definition holds when $y \in \{-1, 1\}$ and $h(\mathbf{w}, \mathbf{x}) \in \mathbb{R}$. There is another, equivalent, definition when we focus on the probability of belonging to one class or the other, *i.e.* when $y \in \{0, 1\}$ and the score $h(\mathbf{w}, \mathbf{x}) \in \mathbb{R}$ is turned into a probability $\sigma(h(\mathbf{w}, \mathbf{x})) \in [0, 1]$ with the logistic function $\sigma(t) = 1/(1 + \exp(-t))$. This variant [Cox \(1958\)](#), also called the *cross-entropy* loss, is defined by

$$\ell(h(\mathbf{w}, \mathbf{x}), y) = -y \ln \left(\frac{1}{1 + \exp(-h(\mathbf{w}, \mathbf{x}))} \right) - (1 - y) \ln \left(1 - \frac{1}{1 + \exp(-h(\mathbf{w}, \mathbf{x}))} \right).$$

One can check that, up to the constant $1/\ln(2)$, both definitions coincide when the label $y \in \{0, 1\}$ is mapped to $2y - 1 \in \{-1, 1\}$: for $y = 1$ the loss is $\ln(1 + \exp(-h(\mathbf{w}, \mathbf{x})))$, and for $y = 0$ it is $\ln(1 + \exp(h(\mathbf{w}, \mathbf{x})))$.

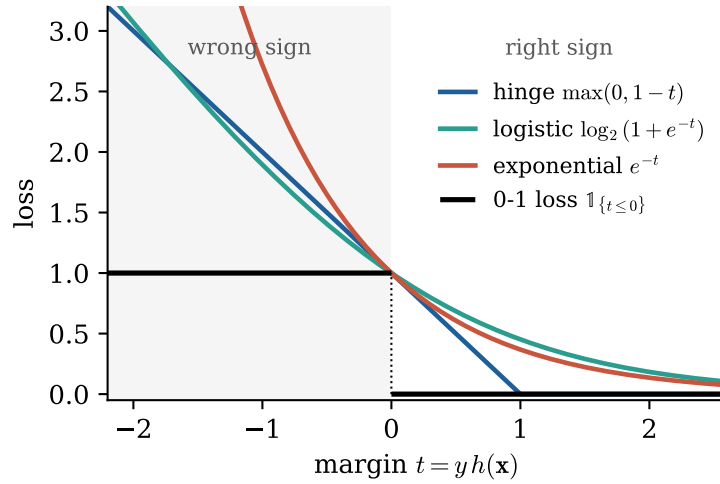


Figure 56: Examples of loss functions used for classification tasks, as functions of the margin $t = yh(\mathbf{x})$: the 0-1 loss, the hinge loss (used by the SVM algorithm), the normalized logistic loss (used for logistic regression) and the exponential loss (used in the context of boosting). All the surrogates upper bound the 0-1 loss and are convex. [code: make_figures_ch07.py: fig_losses]

We now present the main algorithms used in supervised learning.

8.2 k -Nearest Neighbors

A description of the algorithm The k -Nearest Neighbors algorithm (k -NN) [Cover and Hart \(1967\)](#) is a nonparametric method: it does not make any assumption on the underlying distribution of the data. To predict the label of a new instance $\mathbf{x}'$, it computes the “distances” between this new query $\mathbf{x}'$ and all the samples $\mathbf{x}_i \in S$. Then, for a chosen value of k , it looks for the k nearest neighbors of $\mathbf{x}'$ and predicts the label of $\mathbf{x}'$ by a majority vote, *i.e.*

$$\hat{y}(\mathbf{x}') = \arg \max_{y \in \mathcal{Y}} \frac{k_y}{k},$$

where k_y denotes the number of neighbors of $\mathbf{x}'$, among the k nearest ones, whose label is y . In other words, the predicted label $\hat{y}(\mathbf{x}')$ is the most represented class in the k -neighborhood of the new query $\mathbf{x}'$.

Example 8.1. An example of the use of the k -NN algorithm is depicted in Figure 57. It shows how much the prediction for a new instance depends on the choice of k . In this example, the new point is close to points of both classes, which is why its prediction varies with k : it is predicted “red” (class -1) when k is equal to 3 or 5, “blue” (class $+1$) when $k = 9$, and “red” again when $k = 11$.

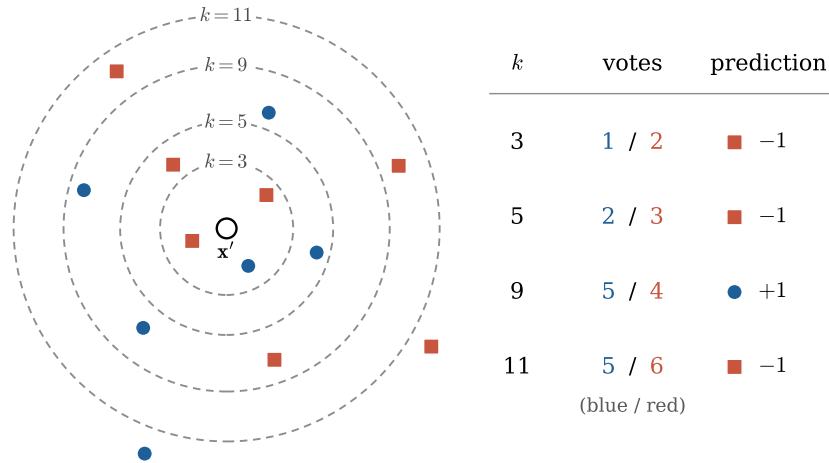


Figure 57: Illustration of the k -NN algorithm with the Euclidean distance. Left: a query $\mathbf{x}'$ (white disc) and the training points of the two classes; the dashed circles contain its k nearest neighbors. Right: number of votes for each class and predicted label. The prediction depends on the value of k .
[code: make_figures_ch07.py: fig_knn_vote]

For small values of k , for instance $k = 1$, the algorithm assigns to the new example $\mathbf{x}'$ the label of its closest example in the training set: similar examples are assumed to share the same label. This rule is the simplest one, and it has been shown [Cover and Hart \(1967\)](#) that, when m tends to infinity, its error rate is at most twice the Bayes error, *i.e.* the smallest error that can be achieved given the distribution of the data. Conversely, when we increase the value of k , the decision becomes smoother. When k is very large (typically when $k = m$), the label given to any new example is simply the majority class of the dataset. This raises an important question: *which value of k should we choose?* In practice, we usually rely on a cross-validation procedure to select the best value of k .

In a nutshell The k -NN rule is a “tell me who your neighbors are” principle, and k is its complexity knob, as shown in Figure 58. With $k = 1$, the decision regions follow every training point, including the noisy ones: the training error is 0 but the boundary is very irregular (low bias, high variance, overfitting). With a very large k , each prediction averages over a large part of the sample: the boundary becomes almost straight and misses the curved structure of the classes (high bias, low variance, underfitting). An intermediate value achieves the best test error. Note that, contrary to most models, the complexity *decreases* when the hyperparameter k increases.

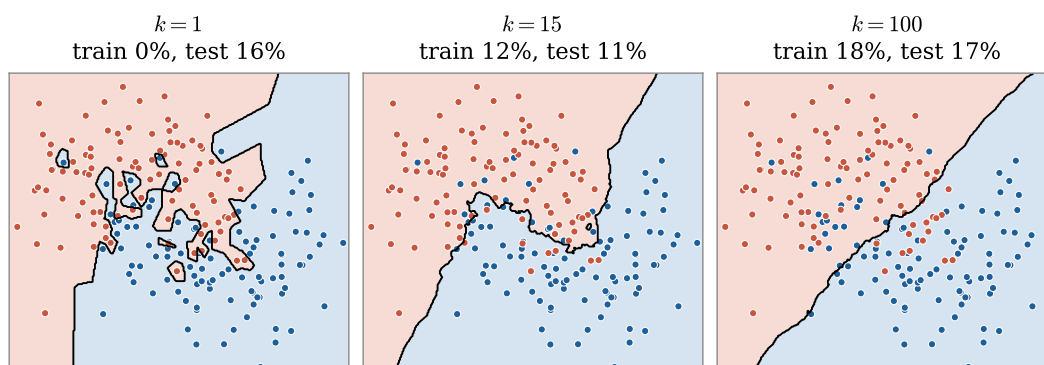


Figure 58: Decision regions of the k -NN rule on a noisy two-moons dataset ($m = 200$) for $k = 1, 15$ and 100 . The training error increases with k , while the test error (estimated on 5000 new points) is the smallest for the intermediate value. [code: make_figures_ch07.py: fig_knn_k]

Some problems The dimension d of the data and the choice of the distance are also important for this algorithm. Indeed, when dealing with high-dimensional data, some features (or attributes), possibly non-informative ones, can have a huge impact on the computation of the distance. This leads to a classification based on irrelevant features and to poor performance. Furthermore, some features can weigh more in the distance computation simply because they take larger values than the other ones. Most of the time, the distance we use is the Euclidean one, *i.e.* for $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$:

$$d_2^2(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2^2 = \sum_{j=1}^d (x_j - x'_j)^2.$$

Example 8.2. Let us consider the following dataset, described by three features:

y	x_1	x_2	x_3
1	0.3	0.89	232
-1	-0.1	0.23	652
-1	0.2	0.7	122
1	0.3	-0.8	232

If we compute the Euclidean distance between the first and the second examples, we get

$$\begin{aligned} d_2^2(\mathbf{x}_1, \mathbf{x}_2) &= \|\mathbf{x}_1 - \mathbf{x}_2\|_2^2 \\ &= \sum_{j=1}^d (x_{1j} - x_{2j})^2 \\ &= (0.3 + 0.1)^2 + (0.89 - 0.23)^2 + (232 - 652)^2 \\ &\simeq (232 - 652)^2. \end{aligned}$$

The first two features have almost no impact on the computed distance, which only depends on the third one.

To overcome this issue, we can change the distance we use. We can, for instance,

use the Manhattan distance (associated with the L_1 -norm)⁹, *i.e.* for $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$:

$$d_1(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_1 = \sum_{j=1}^d |x_j - x'_j|.$$

We can also give a weight w_j to each feature, according to the influence we want it to have. For instance, if a feature is known to be noisy, we can give it a weight close to 0, whereas if it is known to be informative, we can give it a weight close to 1; the weighted Euclidean distance is then $\sqrt{\sum_{j=1}^d w_j(x_j - x'_j)^2}$. The choice of the weights can be guided by domain knowledge.

Finally, another method consists in rescaling the data, so that all the features take their values in the same range or have the same mean and variance. The most common transformations are the following:

- **Standardization (also called normalization or z-score):** for each feature vector $\mathbf{v}_j$, we compute

$$\mathbf{v}_j \leftarrow \frac{\mathbf{v}_j - \mu_j}{\sigma_j},$$

where μ_j and σ_j are respectively the mean and the standard deviation of the feature vector $\mathbf{v}_j$.

- **Rescaling to the hypercube $[-1, 1]^d$:** for each feature vector $\mathbf{v}_j$, we compute

$$\mathbf{v}_j \leftarrow 2 \times \frac{\mathbf{v}_j - \min \mathbf{v}_j}{\max \mathbf{v}_j - \min \mathbf{v}_j} - 1.$$

- **Rescaling to the hypercube $[0, 1]^d$ (also called min-max normalization):** for each feature vector $\mathbf{v}_j$, we compute

$$\mathbf{v}_j \leftarrow \frac{\mathbf{v}_j - \min \mathbf{v}_j}{\max \mathbf{v}_j - \min \mathbf{v}_j}.$$

The first transformation centers and scales the data, so that each feature has zero mean and unit variance, *i.e.* for all $j \in \llbracket 1, d \rrbracket$, the empirical mean of $\mathbf{v}_j$ is 0 and its empirical variance is 1 (note that it does not make the data normally distributed: it only fixes the first two moments). The second and third transformations are very similar: they simply shift and rescale the values of each feature so that they lie in the interval $[-1, 1]$ or $[0, 1]$, respectively.

With the first transformation, all the features have the same variance and thus the same weight in the distance (implicitly, they are all considered as carrying the same quantity of information). This is not the case when we only rescale the features to a hypercube.

⁹Note that all these distances are particular cases of the distance induced by the L_p -norm, defined for all $p \geq 1$ and all $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$ by

$$d_p(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_p = \sqrt[p]{\sum_{j=1}^d |x_j - x'_j|^p}.$$

Algorithm 2: k -Nearest Neighbors

Input: training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$; an example $\mathbf{x}'$ to classify; the number k of nearest neighbors

Output: the predicted label $\hat{y}(\mathbf{x}')$

- 1 **for** $i = 1, \dots, m$ **do**
 - 2 $\perp$ compute the distance $d(\mathbf{x}_i, \mathbf{x}')$;
 - 3 sort the distances in increasing order;
 - 4 keep the k nearest neighbors of $\mathbf{x}'$ in S ;
 - 5 count the number of occurrences of each class among these k neighbors;
 - 6 set $\hat{y}(\mathbf{x}')$ to the most frequent class;
-

Example 8.3. Let us consider the data of Example 8.2 and apply the first transformation (standardization, on the left, using the unbiased estimate of the standard deviation) and the third one (min-max normalization, on the right) to this dataset.

y	x_1	x_2	x_3
1	0.66	0.84	-0.33
-1	-1.45	-0.03	1.46
-1	0.13	0.59	-0.80
1	0.66	-1.40	-0.33

y	x_1	x_2	x_3
1	1	1	0.21
-1	0	0.61	1
-1	0.75	0.89	0
1	1	0	0.21

After either transformation, the three features contribute to the distance on a comparable scale.

Pitfall: forgetting to scale, or scaling with the test set This last suggestion applies to all the algorithms based on a distance or on an inner product (k -NN, SVM, kernel methods, regularized regressions, neural networks): rescaling the data with one of the above methods can be done before using any other learning algorithm. In practice, data normalization or scaling is almost always performed, so keep it in mind when you run experiments. It generally improves the performance of the algorithms when no expert is available to tell us how important the features are. On a practical level, it also makes it easier to compare algorithms with one another.

Be careful, however: the quantities μ_j , σ_j , $\min \mathbf{v}_j$ and $\max \mathbf{v}_j$ must be computed on the *training set only*, and then applied unchanged to the validation and test examples (and, within a cross-validation, recomputed on each training fold). Computing them on the whole dataset lets information about the test set leak into the model and gives over-optimistic performance estimates (see Section 7). Decision trees, on the other hand, are insensitive to such monotone rescalings of the features, since their splits only depend on the order of the values.

Algorithm and complexity The k -NN procedure is described in Algorithm 2. As the reader may notice, to predict the class of a new instance $\mathbf{x}'$, we need to keep the whole training set in memory and to compute the distance between the new example and every training instance.

As a consequence, the prediction of a single example costs $\mathcal{O}(md)$ operations to compute the distances, plus $\mathcal{O}(m \log m)$ operations to sort them (or only $\mathcal{O}(mk)$ if we

Algorithm 3: Condensed Nearest Neighbor

Input: training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$

Output: a smaller sample $S' \subset S$

```
1  $S' \leftarrow \emptyset$ ;  
2 select an example of  $S$  at random, remove it from  $S$  and add it to  $S'$ ;  
3 repeat  
4   foreach example  $(\mathbf{x}_i, y_i) \in S$  do  
5     if  $\mathbf{x}_i$  is misclassified by the 1-NN rule based on  $S'$  then  
6       add  $(\mathbf{x}_i, y_i)$  to  $S'$  and remove it from  $S$ ;  
7     else  
8       keep  $(\mathbf{x}_i, y_i)$  in  $S$ ;  
9 until no example has been added to  $S'$  during a full pass over  $S$ ;
```

simply extract the k smallest distances). Furthermore, the memory required by the algorithm is in $\mathcal{O}(md)$, since the whole training set must be stored. This short analysis shows that this algorithm is not well suited to large datasets.

Pitfall: the curse of dimensionality The k -NN rule relies on the idea that a query has close neighbors in the training set. This idea breaks down when the dimension d grows, a phenomenon called the *curse of dimensionality* [Bellman \(1961\)](#). Consider data uniformly distributed in the unit hypercube $[0, 1]^d$, and a small hypercube around a query containing a fraction r of the data. Its edge length is $r^{1/d}$: to capture only 1% of the data, we need an edge of length $0.01^{1/2} = 0.1$ in dimension 2, but $0.01^{1/10} \simeq 0.63$ in dimension 10 and $0.01^{1/100} \simeq 0.95$ in dimension 100 [Hastie et al. \(2009\)](#). The “neighborhood” then covers almost the whole range of every feature and is no longer local. Moreover, in high dimension, the distances from a query to its nearest and to its farthest neighbors tend to become similar [Beyer et al. \(1999\)](#), so that the notion of nearest neighbor loses its meaning. Feature selection, dimensionality reduction or metric learning are then needed before applying k -NN.

Refinements of the algorithm As we have seen, one drawback of this algorithm is that it needs to store the entire dataset and to compute the distances to all the training samples to predict the label of a new one. However, we can reduce the computation time and the memory footprint with approaches based on the triangle inequality [Elkan \(2003\)](#), on structured partitions of the space such as k -d trees [Bentley \(1975\)](#), or by selecting the most relevant instances of the training set, as in the *Condensed Nearest Neighbor* algorithm [Hart \(1968\)](#) presented in Algorithm 3. This algorithm selects a subset S' of the training set S such that the 1-NN rule based on S' classifies the examples almost as accurately as the 1-NN rule based on the whole dataset S .

Other refinements of the k -NN algorithm exist; the most common one is the weighted k -Nearest Neighbors algorithm [Dudani \(1976\)](#). The idea is simple and consists in assigning a weight to each neighbor in the vote. A standard choice is the inverse of the distance to the query [Liu and Chawla \(2011\)](#): in this way, we give less importance to the examples that are far from the tested one.

Example 8.4. Let us consider an example $\mathbf{x}'$ that we aim to classify with the 3-NN rule, its three nearest neighbors being $(\mathbf{x}_1, y_1)$, $(\mathbf{x}_2, y_2)$ and $(\mathbf{x}_3, y_3)$. Their labels and their distances to $\mathbf{x}'$ are given below:

Example	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$
Label	1	1	0
Distance	3	4	1

If we follow the classical decision rule of this algorithm, we predict that $\mathbf{x}'$ belongs to class 1, since two of its three nearest neighbors belong to this class. This corresponds to the case where every neighbor has the same weight in the decision.

Let us now give each neighbor a weight equal to the inverse of its distance to $\mathbf{x}'$. In this case, the sum of the weights associated with the label 0 is equal to 1, whereas it is only $\frac{1}{3} + \frac{1}{4} = \frac{7}{12} < 1$ for the label 1. Once the weights of the voters are taken into account¹⁰, the new example $\mathbf{x}'$ is therefore given the label 0.

Such a decision rule can be relevant when some regions of the space are sparse and some points are isolated. Another possibility is to learn a new representation of the data, by projecting them into a (better) latent space [He and Wang \(2008\)](#); [Weinberger and Saul \(2009\)](#), as done by Metric Learning algorithms.

An interesting reference [von Luxburg and Bousquet \(2004\)](#) draws a link between the nearest neighbor algorithm (the 1-NN) and the next algorithm we study, the Support Vector Machine (SVM).

8.3 Support Vector Machine

The Support Vector Machine (SVM) is probably one of the best known and most widely used algorithms for binary classification in Machine Learning [Vapnik and Cortes \(1995\)](#). Its derivation relies on the tools of constrained optimization (Lagrangian, dual problem and Karush-Kuhn-Tucker conditions) presented in Section 4.5.

Historical note The idea of a maximal margin linear separator (the “generalized portrait”) goes back to the work of Vapnik and Chervonenkis in the 1960s. It became the modern SVM in two steps: [Boser et al. \(1992\)](#) combined it with the kernel trick to obtain nonlinear classifiers, and [Vapnik and Cortes \(1995\)](#) introduced the soft margin, with slack variables, to handle non-separable data. Together with its theoretical grounding in statistical learning theory, this made kernel SVMs one of the reference methods of the late 1990s and 2000s; see [Scholkopf and Smola \(2001\)](#) for a comprehensive treatment.

Linear SVM The SVM algorithm outputs a hypothesis h which returns the label of an example $\mathbf{x}$. This hypothesis is defined by an (affine) hyperplane which splits the space into two half-spaces, associated with the two labels $\{-1, +1\}$, as follows:

$$h(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) = \begin{cases} -1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b < 0, \\ +1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b > 0. \end{cases} \quad (8.1)$$

¹⁰This notion of (weighted) majority vote is very common in Machine Learning, especially in *PAC-Bayesian learning*. We will see that majority votes are involved in many algorithms, such as boosting, presented in Section 10.

When the two classes can be perfectly separated, there are many hyperplanes, *i.e.* many values of $(\mathbf{w}, b)$, which separate the data well. The idea developed by [Boser et al. \(1992\)](#) is to choose the one with the largest margin. But why? Let us have a look at Figure 59 to understand it.

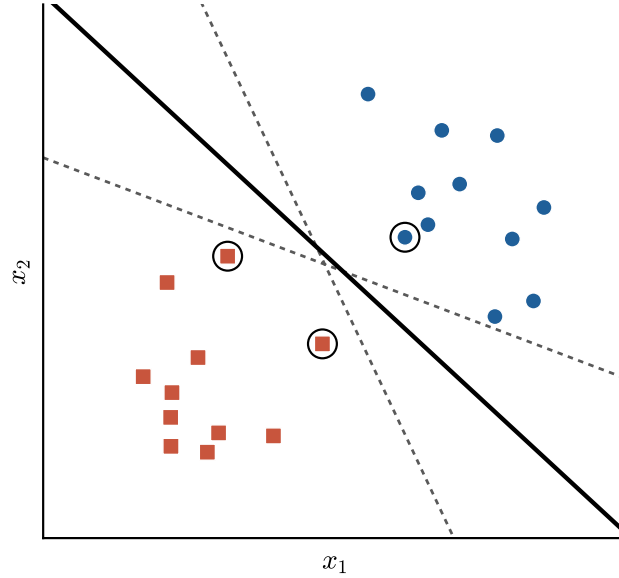


Figure 59: Different linear separators for a classification task. The separator shown as a solid line is the solution of the SVM algorithm, which maximizes the margin (the circled points are the closest ones to it). The two other separators, shown as dashed lines, achieve the same (zero) error on the training data, but pass very close to some training points. [code:

`make_figures_ch07.py: fig_svm_separators]`

To understand why, we simply need to remember our main goal in Machine Learning: finding a classifier which generalizes as well as possible to new data. If we focus on the two dotted lines, we see that they are really close to the examples of one of the classes. At test time, this increases the risk that a new example, drawn from the same distribution, falls on the wrong side of the separator. Thus, to avoid this problem and to select the separator which is likely to generalize best to new data, we choose the one with the largest margin.

In Figure 60, the margin γ is defined as the distance between the two hyperplanes of equations $\langle \mathbf{w}, \mathbf{x} \rangle + b = \pm \rho$. Note that, by rescaling $(\mathbf{w}, b)$, we can always rewrite these equations as $\langle \mathbf{w}, \mathbf{x} \rangle + b = \pm 1$, which we assume from now on. Let us denote $f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$ the score whose sign gives the prediction, and let us consider two points $\mathbf{x}_+$ and $\mathbf{x}_-$ lying respectively on the hyperplanes $f(\mathbf{x}) = +1$ and $f(\mathbf{x}) = -1$ (see Figure 60). Each of these vectors can be decomposed as $\mathbf{x} = \mathbf{x}^{\mathbf{w}} + \mathbf{x}^{\mathbf{w}^\perp}$, where $\mathbf{x}^{\mathbf{w}}$ is collinear with $\mathbf{w}$ and $\mathbf{x}^{\mathbf{w}^\perp}$ is orthogonal to $\mathbf{w}$. This leads to the following relations:

$$\begin{aligned}
 f(\mathbf{x}_+) - f(\mathbf{x}_-) &= 2, \\
 \langle \mathbf{w}, \mathbf{x}_+ \rangle + b - (\langle \mathbf{w}, \mathbf{x}_- \rangle + b) &= 2, \\
 \underbrace{\langle \mathbf{w}, \mathbf{x}_+^{\mathbf{w}} \rangle + \langle \mathbf{w}, \mathbf{x}_+^{\mathbf{w}^\perp} \rangle}_{=0} + b - (\underbrace{\langle \mathbf{w}, \mathbf{x}_-^{\mathbf{w}} \rangle + \langle \mathbf{w}, \mathbf{x}_-^{\mathbf{w}^\perp} \rangle}_{=0} + b) &= 2, \\
 \langle \mathbf{w}, \mathbf{x}_+^{\mathbf{w}} \rangle - \langle \mathbf{w}, \mathbf{x}_-^{\mathbf{w}} \rangle &= 2. \tag{8.2}
 \end{aligned}$$

Furthermore, the margin γ , *i.e.* the distance between the two hyperplanes, is the length of the projection of the vector $\mathbf{x}_+ - \mathbf{x}_-$ onto the unit vector $\frac{\mathbf{w}}{\|\mathbf{w}\|_2}$, *i.e.*

$$\gamma = \frac{\langle \mathbf{x}_+ - \mathbf{x}_-, \mathbf{w} \rangle}{\|\mathbf{w}\|_2}.$$

Thus, we have:

$$\begin{aligned} \gamma &= \frac{\langle \mathbf{x}_+ - \mathbf{x}_-, \mathbf{w} \rangle}{\|\mathbf{w}\|_2} \\ &= \frac{1}{\|\mathbf{w}\|_2} \underbrace{\langle \mathbf{x}_+ - \mathbf{x}_-, \mathbf{w} \rangle} \\ &\quad \downarrow \text{expanding and keeping only the collinear parts} \\ &= \frac{1}{\|\mathbf{w}\|_2} \underbrace{\langle \mathbf{w}, \mathbf{x}_+^{\mathbf{w}} \rangle - \langle \mathbf{w}, \mathbf{x}_-^{\mathbf{w}} \rangle} \\ &\quad \downarrow \text{using Equation (8.2)} \\ &= \frac{2}{\|\mathbf{w}\|_2}. \end{aligned}$$

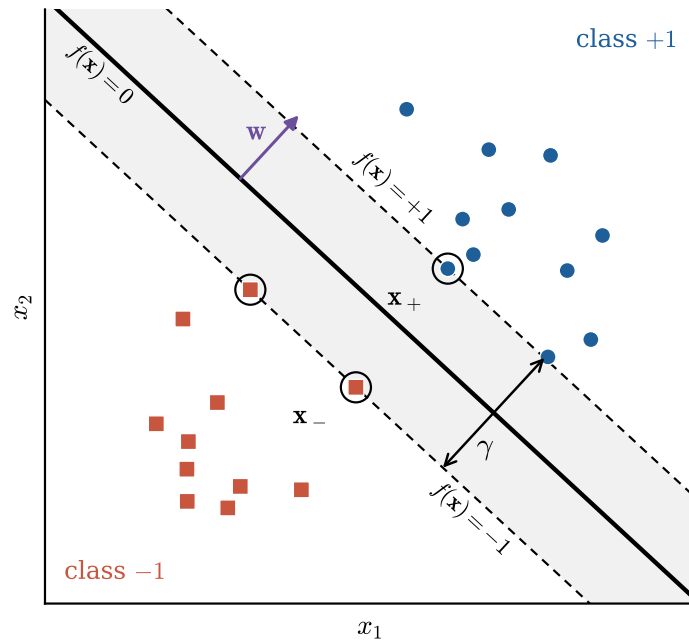


Figure 60: Illustration of the (hard margin) SVM algorithm when the problem is linearly separable, with $f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$. The circled points are the support vectors; $\mathbf{x}_+$ and $\mathbf{x}_-$ lie on the hyperplanes $f(\mathbf{x}) = +1$ and $f(\mathbf{x}) = -1$ (dashed lines), and the distance between these two hyperplanes is the margin $\gamma = 2/\|\mathbf{w}\|_2$. The vector $\mathbf{w}$ is orthogonal to the separator. [code: `make_figures_ch07.py: fig_svm_margin`]

Maximizing the margin γ is thus equivalent to minimizing $\frac{\|\mathbf{w}\|_2}{2}$, or equivalently $\frac{\|\mathbf{w}\|_2^2}{2}$, which is easier to handle since it is differentiable. The minimization problem associated with the *hard margin SVM* is the following.

Definition 8.1: Hard Margin SVM problem

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a sample of size m . The best linear separator in the sense of the SVM algorithm is the solution of the following optimization problem:

$$\begin{aligned} \min_{(\mathbf{w}, b) \in \mathbb{R}^{d+1}} \quad & \frac{1}{2} \|\mathbf{w}\|_2^2 \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1, \quad \text{for all } i = 1, \dots, m, \end{aligned}$$

where the inequality constraints are a compact way of writing that every training example is correctly classified by the rule (8.1), and lies outside the margin.

For the moment, we do not focus on how to solve such an optimization problem (which is a convex constrained optimization problem, more precisely a quadratic program). We will see that, in practice, solvers are available to solve these problems without having to deal with all the mathematical aspects and difficulties.

The problem studied so far corresponds to an ideal situation, where the data are linearly separable. However, in most real cases, the two classes are not linearly separable (see Figure 61), and some points violate the constraint $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1$. This means that such a point is either on the wrong side of the hyperplane, or on the right side of it but inside the margin.

We therefore need a relaxation of the optimization problem which takes this kind of errors into account. These errors are measured by a vector of *slack variables* $\boldsymbol{\xi} = (\xi_1, \dots, \xi_m)$, which is included in the optimization problem. This leads to a new optimization problem, called the *soft margin SVM*, whose aim is to find the right balance between margin maximization and training errors.

Definition 8.2: Soft Margin SVM problem

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a sample of size m and let us denote by $\boldsymbol{\xi} = (\xi_1, \dots, \xi_m)$ the vector of slack variables. The best linear separator in the sense of the SVM algorithm is the solution of the following optimization problem:

$$\begin{aligned} \min_{\boldsymbol{\xi} \in \mathbb{R}^m, (\mathbf{w}, b) \in \mathbb{R}^{d+1}} \quad & \frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{m} \sum_{i=1}^m \xi_i \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \quad \text{for all } i = 1, \dots, m, \\ & \xi_i \geq 0, \quad \text{for all } i = 1, \dots, m. \end{aligned} \tag{8.3}$$

Compared to Definition 8.1, a term $\frac{C}{m} \sum_{i=1}^m \xi_i$ has been added, which represents the cost of violating the constraints. In the first constraint, we allow some violations by subtracting the slack variable ξ_i , which is constrained to be nonnegative: ξ_i measures how much the example $\mathbf{x}_i$ violates the margin constraint. The balance between the two terms of the objective is controlled by a hyperparameter $C > 0$ that has to be tuned during the learning process: the larger C , the more the violations are penalized.

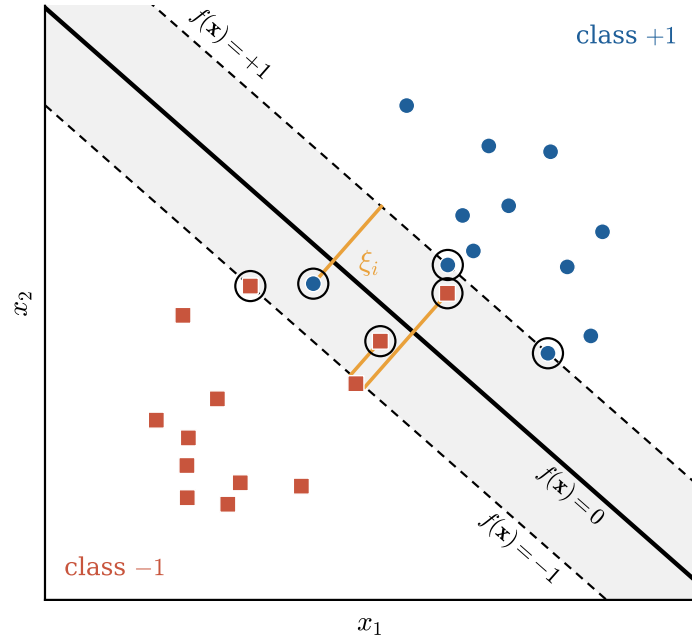


Figure 61: Illustration of the soft margin SVM algorithm when the problem is not linearly separable, with $f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$. The circled points are the support vectors. The dashed lines are the hyperplanes $f(\mathbf{x}) = \pm 1$; the orange segments show the examples that violate the margin constraint, whose slack variable ξ_i is proportional to the length of the segment ($\xi_i > 1$ for a misclassified example).

[code: make_figures_ch07.py: fig_svm_soft]

Proposition 8.1: Equivalent Formulation of the Soft Margin SVM

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a sample of size m . Eliminating the slack variables $\boldsymbol{\xi}$ with the help of the constraints, the optimization problem (8.3) can be rewritten as

$$\min_{(\mathbf{w}, b) \in \mathbb{R}^{d+1}} \frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{m} \sum_{i=1}^m [1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)]_+,$$

where $[x]_+ = \max(0, x)$, *i.e.* the second term is the average hinge loss on S .

Proof. Let us fix $(\mathbf{w}, b)$ and minimize the objective with respect to $\boldsymbol{\xi}$ only. The objective is increasing in each ξ_i , and the two constraints on ξ_i can be summarized as $\xi_i \geq 0$ and $\xi_i \geq 1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)$. Hence, the optimal value of each slack variable is the smallest value satisfying both constraints: it is equal to zero if $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1$, and to the gap $1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)$ otherwise, *i.e.*

$$\forall i = 1, \dots, m, \quad \xi_i = \begin{cases} 0 & \text{if } 1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \leq 0, \\ 1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) & \text{otherwise.} \end{cases}$$

This can be summarized as

$$\forall i = 1, \dots, m, \quad \xi_i = [1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)]_+.$$

Plugging these values into the objective gives the announced problem. $\square$

However, this problem is not so easy to optimize: even though it is convex, it is not smooth (the hinge loss is not differentiable at 1). Furthermore, when the dimension of the data is high, standard (second-order) methods applied to the primal problem have a complexity in $\mathcal{O}(d^3)$ in general [Chapelle \(2007\)](#). We therefore usually solve the so-called *dual formulation* of the problem (see Section 4.5 for the notions of Lagrangian and dual problem): its number of variables is the number m of examples instead of the dimension d , its constraints are very simple, and, as we will see, it only involves the data through inner products.

Proposition 8.2: Dual Problem of the SVM

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a sample of size m . The dual formulation of the soft margin SVM described in Definition 8.2 is

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & -\frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \sum_{i=1}^m \alpha_i \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq \frac{C}{m}, \quad \text{for all } i = 1, \dots, m, \\ & \sum_{i=1}^m y_i \alpha_i = 0. \end{aligned} \quad (8.4)$$

The vector α is the vector of Lagrange multipliers, also called dual variables.

Proof. We are dealing with a constrained optimization problem. To derive its dual formulation, we use the Lagrangian of this problem:

$$\mathcal{L}(\mathbf{w}, b, \xi, \alpha, \beta) = \frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{m} \sum_{i=1}^m \xi_i + \sum_{i=1}^m \alpha_i (1 - \xi_i - y_i (\langle \mathbf{w}, \mathbf{x}_i \rangle + b)) - \sum_{i=1}^m \beta_i \xi_i,$$

where $\alpha \geq 0$ and $\beta \geq 0$ are the vectors of dual variables associated respectively with the first and the second constraints. The dual function is obtained by minimizing $\mathcal{L}$ with respect to the primal variables $(\mathbf{w}, b, \xi)$. Since $\mathcal{L}$ is convex and differentiable in these variables, its minimum, when it is finite, is reached at the points where its gradient vanishes:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}}(\mathbf{w}, b, \xi, \alpha, \beta) = 0, \quad \frac{\partial \mathcal{L}}{\partial b}(\mathbf{w}, b, \xi, \alpha, \beta) = 0, \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \xi}(\mathbf{w}, b, \xi, \alpha, \beta) = 0.$$

These conditions allow us to express the primal variable $\mathbf{w}$ as a function of the dual ones (α and β), and they give constraints on the dual variables. Remember that these stationarity conditions are part of the Karush-Kuhn-Tucker (KKT) conditions (Section 4.5). We respectively have:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}}(\mathbf{w}, b, \xi, \alpha, \beta) = 0 \iff \mathbf{w} - \sum_{i=1}^m y_i \alpha_i \mathbf{x}_i = 0 \iff \mathbf{w} = \sum_{i=1}^m y_i \alpha_i \mathbf{x}_i, \quad (8.5)$$

$$\frac{\partial \mathcal{L}}{\partial b}(\mathbf{w}, b, \xi, \alpha, \beta) = 0 \iff \sum_{i=1}^m \alpha_i y_i = 0, \quad (8.6)$$

$$\frac{\partial \mathcal{L}}{\partial \xi_i}(\mathbf{w}, b, \xi, \alpha, \beta) = 0 \iff \frac{C}{m} - \alpha_i - \beta_i = 0 \iff \beta_i = \frac{C}{m} - \alpha_i. \quad (8.7)$$

(If (8.6) or (8.7) is not satisfied, $\mathcal{L}$ is affine and non-constant in b or in ξ_i , and its infimum is $-\infty$.) We now plug these results into the expression of the Lagrangian:

$$\begin{aligned}
\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \beta) &= \underbrace{\frac{1}{2} \|\mathbf{w}\|_2^2}_{} + \frac{C}{m} \sum_{i=1}^m \xi_i + \sum_{i=1}^m \alpha_i \left(1 - \xi_i - y_i \left(\underbrace{\langle \mathbf{w}, \mathbf{x}_i \rangle}_{\text{red}} + b \right) \right) - \sum_{i=1}^m \beta_i \xi_i \\
&\downarrow \text{ using Equation (8.5) and rearranging the terms} \\
&= \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \sum_{i=1}^m \xi_i \left(\underbrace{\frac{C}{m} - \alpha_i - \beta_i}_{\text{green}} \right) + \sum_{i=1}^m \alpha_i \\
&\quad - \sum_{i=1}^m \alpha_i y_i \underbrace{\left\langle \sum_{j=1}^m y_j \alpha_j \mathbf{x}_j, \mathbf{x}_i \right\rangle}_{\text{red}} - b \underbrace{\sum_{i=1}^m \alpha_i y_i}_{\text{orange}} \\
&\downarrow \text{ using Equations (8.6) and (8.7)} \\
&= \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \sum_{i=1}^m \xi_i \times \mathbf{0} + \sum_{i=1}^m \alpha_i \\
&\quad - \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_j, \mathbf{x}_i \rangle - b \times \mathbf{0} \\
&\downarrow \text{ computing the difference} \\
&= -\frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \sum_{i=1}^m \alpha_i.
\end{aligned}$$

The dual problem consists in maximizing this dual function with respect to the dual variables. Finally, the constraints of the dual problem come from Equation (8.6), and from Equation (8.7) combined with the nonnegativity of the multipliers: $\alpha_i \geq 0$ and $\beta_i = \frac{C}{m} - \alpha_i \geq 0$, i.e. $0 \leq \alpha_i \leq \frac{C}{m}$. $\square$

Note that the dual problem is a concave maximization problem with respect to the dual variables (a convex quadratic program). Indeed, the constraints are linear, hence define a convex set. As for the objective function, it can be rewritten using a quadratic form:

$$\max_{\boldsymbol{\alpha}} -\frac{1}{2} \boldsymbol{\alpha}^T \mathbf{G} \boldsymbol{\alpha} + \sum_{i=1}^m \alpha_i,$$

where $\mathbf{G}$ is the matrix defined by $G_{ij} = y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle$. This matrix is the Gram matrix of the vectors $y_1 \mathbf{x}_1, \dots, y_m \mathbf{x}_m$, and it is therefore positive semi-definite: $\boldsymbol{\alpha}^T \mathbf{G} \boldsymbol{\alpha} = \|\sum_{i=1}^m \alpha_i y_i \mathbf{x}_i\|_2^2 \geq 0$. Moreover, since the primal problem is convex with affine constraints and is feasible, strong duality holds (see Section 4.5): the optimal values of the primal and dual problems coincide. Note however that $\mathbf{G}$ is in general not positive definite (it has rank at most d), so the dual objective is not *strictly* concave and the optimal $\boldsymbol{\alpha}$ need not be unique; the optimal vector $\mathbf{w}$, on the other hand, is unique since the primal objective is strictly convex in $\mathbf{w}$.

Remarks. The dual problem looks for a solution in a different space: we work in the space of the examples (one variable α_i per example) rather than in the space of the parameters. Furthermore, once the dual problem is solved, Equation (8.5) gives the primal solution $\mathbf{w}$. The value of the offset b does not appear in the stationarity conditions, but it can be recovered from the complementary slackness conditions, as we will see below.

Although the dual formulation leads to a problem that is easier to solve when the number of examples is small, it does not solve the problem of data that are not linearly separable. To tackle this issue, we use what is commonly called the *kernel trick*.

In a nutshell Consider the one-dimensional data of Figure 62: the positive examples lie on both sides of the negative ones, so no threshold on x , *i.e.* no linear classifier in $\mathbb{R}$, can separate them. If we map each example to $\Phi(x) = (x, x^2) \in \mathbb{R}^2$, the two classes become separable by the horizontal line $x^2 = c$, *i.e.* by the linear classifier of $\mathbb{R}^2$ with $\mathbf{w} = (0, 1)$ and $b = -c$. Back in the input space, this linear classifier is the *nonlinear* rule “predict +1 if $|x| > \sqrt{c}$ ”. The SVM only needs the inner products $\langle \Phi(x), \Phi(x') \rangle = xx' + x^2x'^2$, which can be computed directly from x and x' : this function is a kernel, and the map Φ never has to be computed explicitly.

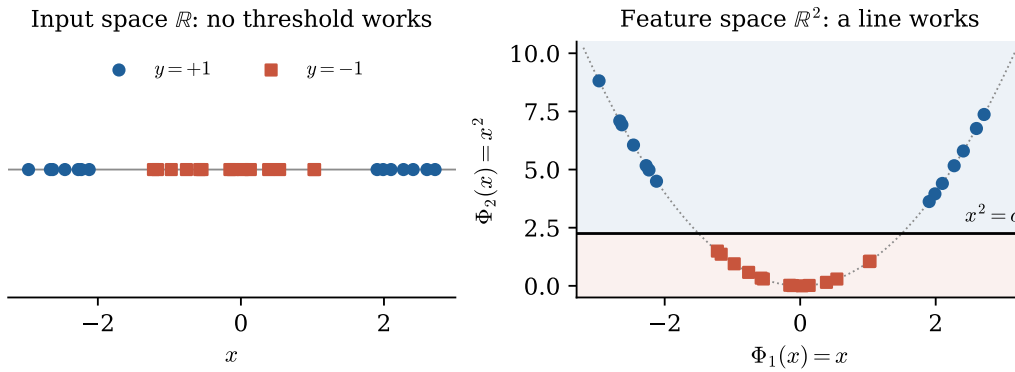


Figure 62: The idea behind the kernel trick. Left: one-dimensional data that are not linearly separable. Right: after the mapping $\Phi(x) = (x, x^2)$, a line separates the two classes; it corresponds to the nonlinear rule $|x| > \sqrt{c}$ in the input space. [code: make_figures_ch07.py: fig_kernel_lift]

Kernel SVM Instead of using the standard inner product between two examples, we define a function $K(\cdot, \cdot)$ which takes two vectors as input and returns a real number. Such a function K is called a (positive semi-definite) kernel when it is (i) symmetric and (ii) positive semi-definite, in the sense that every matrix $\mathbf{K}$ built from it on a finite sample is positive semi-definite:

- (i) for all $(\mathbf{x}, \mathbf{x}') \in \mathbb{R}^d \times \mathbb{R}^d$, we have $K(\mathbf{x}, \mathbf{x}') = K(\mathbf{x}', \mathbf{x})$;
- (ii) for all $m \geq 1$, all $\mathbf{x}_1, \dots, \mathbf{x}_m \in \mathbb{R}^d$ and all $\mathbf{c} \in \mathbb{R}^m$, we have $\mathbf{c}^T \mathbf{K} \mathbf{c} = \sum_{i=1}^m \sum_{j=1}^m c_i c_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0$, where $\mathbf{K}$ is the matrix of general term $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$.

These properties of the function K (or of the matrices $\mathbf{K}$) play a key role and lead to the following result due to Mercer [Mercer \(1909\)](#).

Theorem 8.3: Mercer Theorem

Let $\mathcal{X}$ be a compact subset of $\mathbb{R}^d$ and let K be a continuous, symmetric, positive semi-definite function, *i.e.* a kernel. Then, there exist an orthonormal family of functions $(\Phi_j)_{j \geq 1}$ of $L^2(\mathcal{X})$ and a sequence $(\lambda_j)_{j \geq 1}$ of nonnegative real numbers such that

$$K(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{\infty} \lambda_j \Phi_j(\mathbf{x}) \Phi_j(\mathbf{x}') = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle,$$

where $\Phi(\mathbf{x}) = (\sqrt{\lambda_1} \Phi_1(\mathbf{x}), \dots, \sqrt{\lambda_j} \Phi_j(\mathbf{x}), \dots)$ is the representation of $\mathbf{x}$ in a new space.

The kernel trick thus says that there is no need to know the transformation Φ explicitly: the knowledge of K is enough, since the dual problem and the predictions only involve inner products between examples. Furthermore, it makes it possible to (implicitly) project the data into a higher-dimensional space, called the feature space (possibly of infinite dimension), in which the classes may become linearly separable.

Introducing the kernel K into the optimization problem (8.4) leads to the following dual formulation:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & -\frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^m \alpha_i \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq \frac{C}{m}, \quad \text{for all } i = 1, \dots, m, \\ & \sum_{i=1}^m y_i \alpha_i = 0. \end{aligned} \quad (8.8)$$

There exists a large number of kernel functions (see [Genton \(2002\)](#) for a more complete list), among which the most widely used are:

- **Linear kernel:** $K(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$, which is the standard inner product.
- **Polynomial kernel:** $K(\mathbf{x}, \mathbf{x}') = (\langle \mathbf{x}, \mathbf{x}' \rangle + c)^p$, where $c \geq 0$ is a constant and $p \in \mathbb{N}^*$ is the degree.
- **Gaussian kernel:** $K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\sigma^2}\right)$, where $\sigma > 0$ is a hyperparameter that needs to be tuned. It controls how fast the similarity between two instances decreases with their distance. The larger σ , the slower this decrease: distant examples are then still considered as similar, and all the examples of the dataset play a more uniform role, which leads to smoother decision boundaries.

Example 8.5. Let us consider two 2-dimensional vectors $\mathbf{x}$ and $\mathbf{z}$ and the polynomial kernel of degree 2. We aim to write this kernel as an inner product, *i.e.* to find a mapping function Φ such that $K(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$:

$$\begin{aligned} K(\mathbf{x}, \mathbf{z}) &= (x_1 z_1 + x_2 z_2 + c)^2 \\ &= c^2 + x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 x_2 z_2 + 2cx_1 z_1 + 2cx_2 z_2 \\ &= \left(c, x_1^2, x_2^2, \sqrt{2}x_1 x_2, \sqrt{2}cx_1, \sqrt{2}cx_2\right) \left(c, z_1^2, z_2^2, \sqrt{2}z_1 z_2, \sqrt{2}cz_1, \sqrt{2}cz_2\right)^T \end{aligned}$$

$$= \Phi(\mathbf{x})^T \Phi(\mathbf{z}),$$

where $\Phi : \mathbf{x} \mapsto (c, x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2}cx_1, \sqrt{2}cx_2)$. Using this kernel, we thus implicitly project the data into a 6-dimensional space.

As an exercise, apply the same process to show that, for a 3-dimensional dataset and a polynomial kernel of degree 2 with $c = 0$, the data are projected into a space of dimension 6.

Let us now show why the Gaussian kernel projects the data into a space of infinite dimension. Let us consider two instances $\mathbf{x}$ and $\mathbf{z}$ in $\mathbb{R}^d$ and recall that

$$\|\mathbf{x} - \mathbf{z}\|_2^2 = \|\mathbf{x}\|_2^2 + \|\mathbf{z}\|_2^2 - 2\langle \mathbf{x}, \mathbf{z} \rangle.$$

Thus, we can write¹¹

$$\begin{aligned} K(\mathbf{x}, \mathbf{z}) &= \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|_2^2}{2\sigma^2}\right) \\ &\quad \downarrow \text{using the previous equation and the properties of the exponential} \\ &= \underbrace{\exp\left(-\frac{\|\mathbf{x}\|_2^2}{2\sigma^2}\right)}_{c(\mathbf{x})} \underbrace{\exp\left(-\frac{\|\mathbf{z}\|_2^2}{2\sigma^2}\right)}_{c(\mathbf{z})} \exp\left(\frac{\langle \mathbf{x}, \mathbf{z} \rangle}{\sigma^2}\right) \\ &\quad \downarrow \text{using the power series expansion of the exponential function} \\ &= c(\mathbf{x})c(\mathbf{z}) \sum_{k=0}^{\infty} \frac{\langle \mathbf{x}, \mathbf{z} \rangle^k}{k! \sigma^{2k}} \\ &\quad \downarrow \text{writing } \langle \mathbf{x}, \mathbf{z} \rangle^k = \langle \mathbf{x}^{\otimes k}, \mathbf{z}^{\otimes k} \rangle, \text{ where } \mathbf{x}^{\otimes k} \in \mathbb{R}^{d^k} \text{ gathers all the products } x_{j_1} \cdots x_{j_k} \\ &= \sum_{k=0}^{\infty} \left\langle \frac{c(\mathbf{x})}{\sqrt{k!} \sigma^k} \mathbf{x}^{\otimes k}, \frac{c(\mathbf{z})}{\sqrt{k!} \sigma^k} \mathbf{z}^{\otimes k} \right\rangle. \end{aligned}$$

Hence $K(\mathbf{x}, \mathbf{z}) = \langle \Phi(\mathbf{x}), \Phi(\mathbf{z}) \rangle$, where $\Phi(\mathbf{x})$ is the infinite sequence obtained by concatenating the vectors $\frac{c(\mathbf{x})}{\sqrt{k!} \sigma^k} \mathbf{x}^{\otimes k}$ for $k = 0, 1, 2, \dots$: the feature space associated with the Gaussian kernel contains all the monomials of all degrees, and is thus of infinite dimension.

Figure 63 illustrates the use of a linear and of a Gaussian kernel. Do you think that a polynomial kernel of degree 2 would be enough, in this case, to separate the two classes?

Tuning the Gaussian kernel SVM A Gaussian kernel SVM has two hyperparameters, which must be tuned jointly (typically by a grid search with cross-validation, on a logarithmic grid). In software such as `scikit-learn`, the Gaussian kernel is parameterized as $K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|_2^2)$, *i.e.* $\gamma = 1/(2\sigma^2)$, and the cost parameter multiplies the *sum* of the slack variables, so that it corresponds to our C/m . Figure 64 shows their effect:

- a large γ (small σ) makes the similarity very local: each example only influences its immediate neighborhood, and the boundary can wrap around individual points (overfitting); a small γ gives a smooth, almost linear, boundary;

¹¹See the development [here](#) for further details.

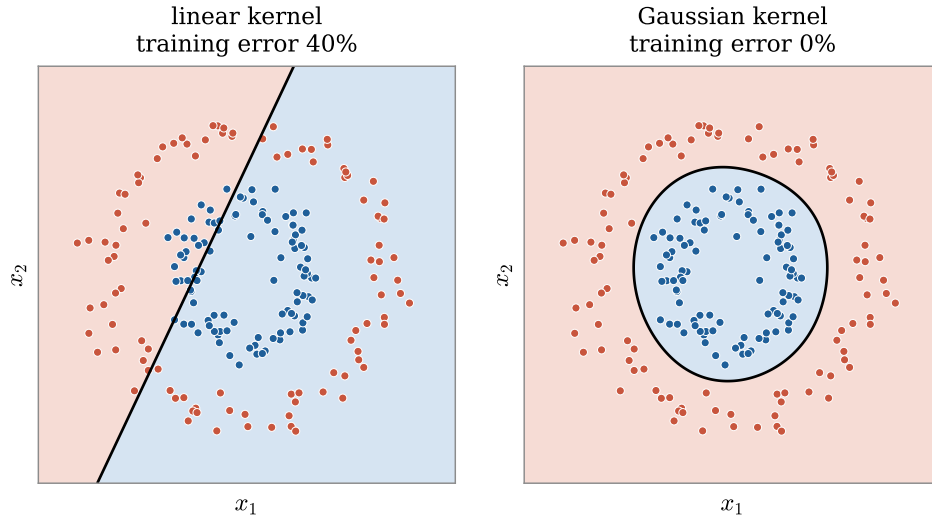


Figure 63: Illustration of the kernel SVM algorithm when the problem is not linearly separable (linear kernel on the left, Gaussian kernel on the right). Blue points represent the positive class and red points the negative class. The colored areas represent the decision regions of the learned hypothesis.

[code: make_figures_ch07.py: fig_kernel_svm]

- a small C favors a large margin at the price of many margin violations, so that many examples are support vectors; a large C forces the classifier to fit the training data more closely.

Prediction phase With the linear SVM (hard or soft margin), to predict the label of a new instance $\mathbf{x}'$, we compute

$$h(\mathbf{x}') = \text{sign}(\langle \mathbf{w}, \mathbf{x}' \rangle + b).$$

With the kernelized version, using Equation (8.5) in the feature space, the hypothesis (or classifier) takes the following form:

$$h(\mathbf{x}') = \text{sign} \left(\sum_{i=1}^m \alpha_i y_i K(\mathbf{x}', \mathbf{x}_i) + b \right).$$

Note that, in this case, we need to compute the similarity between the new example and the training examples. However, both hypotheses are linear: they simply work in different spaces (the input space for the former, the feature space for the latter).

A more detailed analysis of the dual formulation We now take advantage of the dual formulation presented above to study its link with the primal solution, and to explain why this model is called a *Support Vector Machine*.

For the first point, we write the full set of KKT conditions (please refer to Section 4.5, and more precisely to Section 4.5, for further details on how to obtain them) for the soft margin SVM problem (see Definition 8.2), using the previous notation:

- Stationarity conditions:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = 0 \iff \mathbf{w} = \sum_{i=1}^m y_i \alpha_i \mathbf{x}_i, \quad (8.9)$$

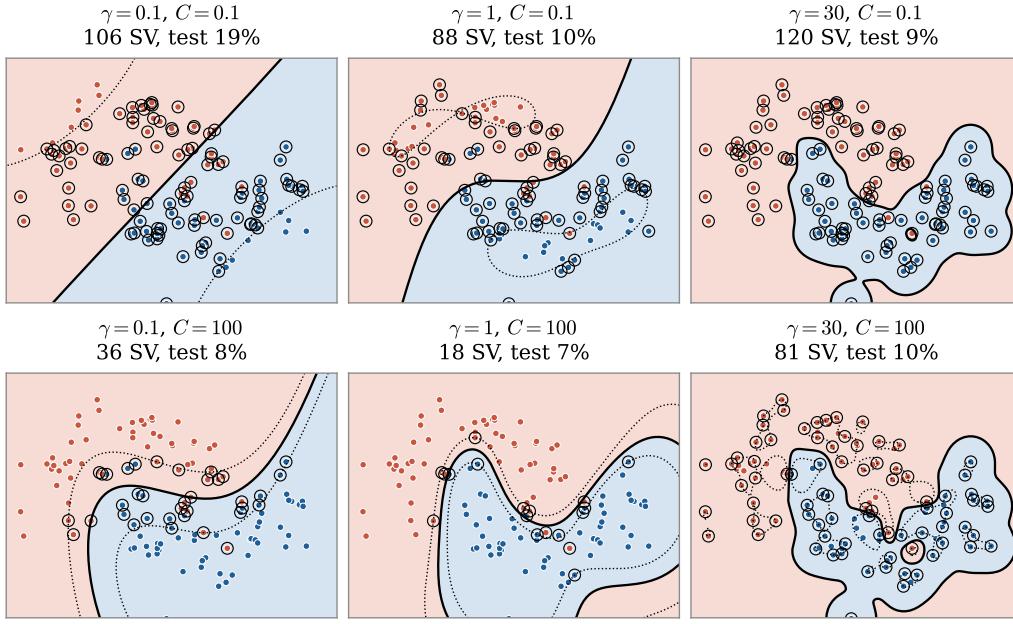


Figure 64: Gaussian kernel SVM (`scikit-learn` parameterization) on a two-moons dataset ($m = 120$), for three bandwidths γ (columns) and two costs C (rows). Solid line: decision boundary $f(\mathbf{x}) = 0$; dotted lines: $f(\mathbf{x}) = \pm 1$; circled points: support vectors. The titles give the number of support vectors and the test error. [code: `make_figures_ch07.py` fig_svm_rbf]

$$\frac{\partial \mathcal{L}}{\partial b}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = 0 \iff \sum_{i=1}^m \alpha_i y_i = 0, \quad (8.10)$$

$$\frac{\partial \mathcal{L}}{\partial \xi_i}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = 0 \iff \alpha_i + \beta_i = \frac{C}{m}, \quad \forall i = 1, \dots, m. \quad (8.11)$$

- Primal feasibility:

$$1 - \xi_i - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \leq 0, \quad \forall i = 1, \dots, m, \quad (8.12)$$

$$\xi_i \geq 0, \quad \forall i = 1, \dots, m. \quad (8.13)$$

- Dual feasibility:

$$\alpha_i \geq 0, \quad \forall i = 1, \dots, m, \quad (8.14)$$

$$\beta_i \geq 0, \quad \forall i = 1, \dots, m. \quad (8.15)$$

- Complementary slackness:

$$\alpha_i (1 - \xi_i - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)) = 0, \quad \forall i = 1, \dots, m, \quad (8.16)$$

$$\beta_i \xi_i = 0, \quad \forall i = 1, \dots, m. \quad (8.17)$$

Equation (8.9) immediately gives an expression of $\mathbf{w}$ in terms of the dual variables. It remains to find an expression of the offset b to fully recover the primal solution. The stationarity conditions do not provide it, but, as we will see, the complementary slackness conditions do.

Let us now focus on the *support vectors*, i.e. the training examples $\mathbf{x}_i$ for which $\alpha_i > 0$. We will see that they are points which lie on the margin, inside it, or on the

wrong side of the hyperplane (the circled points in Figures 60 and 61), and that they are the only points needed to define our separator. We will also describe the role of all the other points.

To do so, we focus on the complementary slackness conditions (8.16) and (8.17), which hold for every instance $i = 1, \dots, m$. Since $0 \leq \alpha_i \leq \frac{C}{m}$ for all $i = 1, \dots, m$, there are three possible cases for α_i :

- i) if $\alpha_i = 0$, then Equation (8.11) implies $\beta_i = \frac{C}{m} > 0$, and thus $\xi_i = 0$ by Equation (8.17). Using the primal feasibility condition (8.12), we then get $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1$. So the point $\mathbf{x}_i$ is **correctly classified and lies outside the margin** (or on its boundary).
- ii) if $\alpha_i = \frac{C}{m}$, then Equation (8.11) implies $\beta_i = 0$, so Equation (8.17) only tells us that $\xi_i \geq 0$. Equation (8.16) then gives $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = 1 - \xi_i \leq 1$. So the point $\mathbf{x}_i$ lies **on the margin** (if $\xi_i = 0$), **inside the margin** (if $0 < \xi_i \leq 1$), or is **misclassified** (if $\xi_i > 1$).
- iii) finally, if $0 < \alpha_i < \frac{C}{m}$, then Equation (8.11) implies $0 < \beta_i < \frac{C}{m}$, and thus $\xi_i = 0$ by Equation (8.17). Equation (8.16) then gives $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = 1$. So the point $\mathbf{x}_i$ lies **exactly on the margin**.

The last case gives us the value of the offset: for any index i such that $0 < \alpha_i < \frac{C}{m}$, since $y_i \in \{-1, 1\}$, we have

$$b = y_i - \langle \mathbf{w}, \mathbf{x}_i \rangle = y_i - \sum_{j=1}^m \alpha_j y_j \langle \mathbf{x}_j, \mathbf{x}_i \rangle,$$

where the inner product is replaced by $K(\mathbf{x}_j, \mathbf{x}_i)$ in the kernel case. In practice, to obtain a numerically more stable value, we average this quantity over all the support vectors lying exactly on the margin.

If we combine this analysis with the fact that $\mathbf{w} = \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i$, we note that the points lying strictly outside the margin ($\alpha_i = 0$) do not contribute to the definition of the hyperplane at all! Only the support vectors matter, hence the name of the algorithm. This is important from a computational point of view: we do not need to use all the training points in the prediction phase, *i.e.* there is no need to compute the similarities between a new example and all the training examples, but only with the support vectors.

It is also possible to speed up the learning phase of the (kernel) SVM, for instance by using a low-rank approximation of the kernel matrix; we do not go into these details here.

The next algorithm we study can be used for both regression and classification tasks, depending on the nature of the output.

8.4 Linear Regression and Logistic Regression

The support vector machine was a first example of a linear model. In this section, we present the well-known linear regression model and its counterpart for classification tasks, logistic regression.

Linear Regression Linear regression is mainly used when we aim to predict a real value, *e.g.* the price of a house or the score obtained at a given exam, as illustrated in Figure 65.

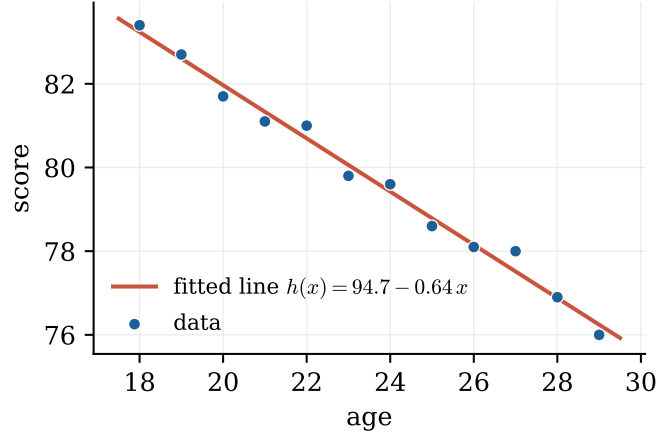


Figure 65: Example of a linear regression, where the aim is to predict the score of a person according to his or her age. The solution obtained by linear regression is represented in red.

[code: make_figures_ch07.py: fig_linear_score]

As for the linear SVM, the learned model is affine: in its simplest form (a single input variable), it is a straight line, and a hyperplane in general. Given $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m \subset \mathbb{R}^d \times \mathbb{R}$, we aim to find the hyperplane which best approximates the scatter plot, *i.e.* to learn a hypothesis h of the form

$$h(\boldsymbol{\theta}, \mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_d x_d.$$

To learn the parameters $\boldsymbol{\theta} = (\theta_0, \dots, \theta_d)$ of the model, we minimize a loss, as for the SVM. For regression tasks, as stated in Section 7.2, we usually minimize the Mean Squared Error (MSE).

Proposition 8.4: Linear Regression

We consider the following probabilistic model for our data:

$$Y = \theta_0 + \theta_1 X_1 + \dots + \theta_d X_d + \varepsilon,$$

where Y is the variable to predict, $X = (X_1, \dots, X_d)$ is the vector of variables used for the prediction, and ε is a centered random noise representing the error of the model. We consider a hypothesis h of the form

$$h(\boldsymbol{\theta}, \mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_d x_d.$$

Given a set S of m examples, let $\mathbf{X} \in \mathbb{R}^{m \times (d+1)}$ be the design matrix whose i -th row is $(1, \mathbf{x}_i^T)$, and let $\mathbf{y} = (y_1, y_2, \dots, y_m)^T$. If $\mathbf{X}^T \mathbf{X}$ is invertible, then the solution of the Mean Squared Error problem

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^{d+1}} \|\mathbf{y} - h(\boldsymbol{\theta}, \mathbf{X})\|_2^2 = \min_{\boldsymbol{\theta} \in \mathbb{R}^{d+1}} \sum_{i=1}^m (y_i - h(\boldsymbol{\theta}, \mathbf{x}_i))^2,$$

where $h(\boldsymbol{\theta}, \mathbf{X}) = \mathbf{X}\boldsymbol{\theta}$ is the vector of predictions, is given by

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}.$$

Proof. The objective function $F(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$ is convex, since its Hessian matrix $2\mathbf{X}^T\mathbf{X}$ is positive semi-definite. Its minimizers are thus the points where its gradient vanishes:

$$\nabla F(\boldsymbol{\theta}) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) = 0 \iff \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} = \mathbf{X}^T\mathbf{y},$$

and these *normal equations* have the unique solution $\hat{\boldsymbol{\theta}}$ when $\mathbf{X}^T\mathbf{X}$ is invertible. This is exactly the proof studied in the course on linear models. $\square$

This first model is quite simple, and an analytical solution is available: no iterative optimization procedure is needed to learn the model¹².

Let us now place ourselves in a slightly different context and consider the following example.

Example 8.6. We are trying to build a regression model able to determine whether or not an individual has an infection, according to his or her lymphocyte count. The variable to predict can take two values: 1 if the person has an infection and 0 otherwise.

At first sight, nothing prevents us from learning a linear model, by least squares, to fit this new point cloud, as illustrated in panel (a) of Figure 66. We then simply have to choose a threshold on the values taken by the hypothesis h , above which an individual is considered as sick: *e.g.* we consider that an example $\mathbf{x}$ belongs to the positive class when $h(\mathbf{x})$ is greater than 0.5 (*i.e.* the prediction is negative on the left of the vertical dotted line and positive on its right). In this example, it works well.

Let us now consider another dataset, in which the number of lymphocytes can be extremely high, meaning that the infection is serious. This new dataset is represented in panel (b) of Figure 66. This time, we see that if we use the same threshold, we miss some positive instances, *i.e.* some infected people: the few very large values have “pulled” the line and shifted the threshold to the right (from 3 to about 4.4), so that the patient with a count of 4 is now predicted healthy.

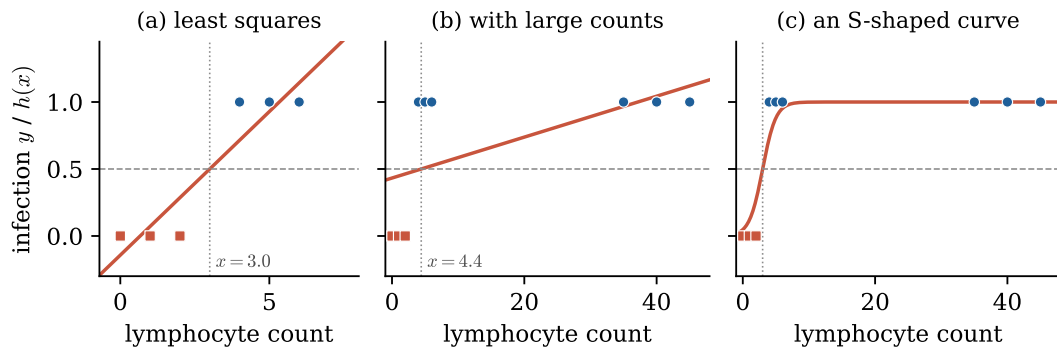


Figure 66: Predicting an infection ($y = 1$, blue) from the lymphocyte count. (a) A least squares line and the threshold $h(x) = 0.5$ separate the two classes. (b) Adding a few patients with very large counts pulls the line and moves the threshold to the right: a sick patient is now misclassified. (c) An S-shaped curve with values in $[0, 1]$ is not affected by these extreme values.

[code: make_figures_ch07.py: fig_logistic_motivation]

This example shows that the linear model is not well suited to this problem: we need a different shape, *i.e.* a curve which is better adapted to the structure of our data,

¹²For more information on linear regression, and especially on its statistical aspects, the reader can consult the course on Linear Models given by Stéphane Chrétien, available [here](#).

such as the S-shaped curve of panel (c) of Figure 66. Such a model takes its values in $[0, 1]$, and we can thus interpret it as an estimate of the probability of having an infection. To transform the values predicted by a linear model into probabilities, we use the logistic function, *i.e.* we compute

$$\frac{1}{1 + \exp(-h(\mathbf{x}))}.$$

We then talk about *(linear) logistic regression*.

Logistic Regression The logistic regression model, also called the *logit model*, was introduced in the middle of the 20th century Cox (1958), but the use of *logit* functions dates back to the 19th century Cramer (2003).

This model is close to the SVM in the sense that it also learns a linear separator between two classes. However, it differs from the SVM in that it estimates the probability that an example belongs to a given class, for instance the positive class: $\eta(\mathbf{x}) = \Pr(Y = 1 \mid X = \mathbf{x})$. More precisely, logistic regression models the logarithm of the *odds*, *i.e.* of the ratio between the probabilities of the two classes, with a linear model:

$$\ln \left(\frac{\Pr(y = 1 \mid \mathbf{x})}{\Pr(y = 0 \mid \mathbf{x})} \right) = h(\mathbf{w}, b, \mathbf{x}) = b + \langle \mathbf{w}, \mathbf{x} \rangle.$$

Thus, once the parameters of the model are learned, we can compute the probability of belonging to class 1:

$$\Pr(y = 1 \mid \mathbf{x}) = \frac{\exp(h(\mathbf{w}, b, \mathbf{x}))}{1 + \exp(h(\mathbf{w}, b, \mathbf{x}))} = \frac{1}{1 + \exp(-h(\mathbf{w}, b, \mathbf{x}))}.$$

This function of h is called the *logistic function* (or sigmoid function), and it takes its values in $[0, 1]$. An example $\mathbf{x}$ is (usually) assigned to class 1 if $\Pr(y = 1 \mid \mathbf{x}) > 0.5$, *i.e.* if $h(\mathbf{w}, b, \mathbf{x}) > 0$. Depending on the task and on the objective, we can choose to modify this threshold, as we will see later.

To estimate the parameters of the model, we maximize the likelihood $\mathfrak{L}(\mathbf{w}, b, S)$ of the data, where S is a set of m examples with labels $y_i \in \{0, 1\}$, assumed to be independent:

$$\begin{aligned} \mathfrak{L}(\mathbf{w}, b, S) &= \prod_{i=1}^m \Pr(Y = y_i \mid X = \mathbf{x}_i) \\ &\quad \downarrow \text{separating the examples such that } y_i = 1 \text{ and those such that } y_i = 0 \\ &= \prod_{i: y_i=1} \Pr(Y = 1 \mid X = \mathbf{x}_i) \times \prod_{i: y_i=0} \Pr(Y = 0 \mid X = \mathbf{x}_i) \\ &\quad \downarrow \text{using the fact that, given } \mathbf{x}_i, \text{ the label follows a Bernoulli distribution} \\ &= \prod_{i=1}^m \left(\frac{1}{1 + \exp(-h(\mathbf{w}, b, \mathbf{x}_i))} \right)^{y_i} \times \left(\frac{1}{1 + \exp(h(\mathbf{w}, b, \mathbf{x}_i))} \right)^{1-y_i}. \end{aligned}$$

We usually prefer to minimize the negative log-likelihood of the data, which turns the product into a sum:

$$\ell(\mathbf{w}, b, S) = -\ln(\mathfrak{L}(\mathbf{w}, b, S))$$

$$= - \sum_{i=1}^m \left[y_i \ln \left(\frac{1}{1 + \exp(-(\langle \mathbf{w}, \mathbf{x}_i \rangle + b))} \right) + (1 - y_i) \ln \left(1 - \frac{1}{1 + \exp(-(\langle \mathbf{w}, \mathbf{x}_i \rangle + b))} \right) \right].$$

In this way, we recover the logistic loss (in its version with labels in $\{0, 1\}$) introduced at the beginning of this section. In the following, for the sake of simplicity, we set $g(\mathbf{w}, b, \mathbf{x}) = \frac{1}{1 + \exp(-(\langle \mathbf{w}, \mathbf{x} \rangle + b))}$. The optimization problem then becomes

$$\min_{(\mathbf{w}, b) \in \mathbb{R}^{d+1}} - \frac{1}{m} \sum_{i=1}^m [y_i \ln(g(\mathbf{w}, b, \mathbf{x}_i)) + (1 - y_i) \ln(1 - g(\mathbf{w}, b, \mathbf{x}_i))].$$

We divide the loss by m in order to be consistent with the notion of empirical risk defined previously; this does not change the minimizer.

SVMs and logistic regression models are similar from a geometric point of view (they both learn a hyperplane) and both minimize a (regularized) empirical risk with a convex surrogate of the 0-1 loss. A closer look at their loss functions (Figure 56) nevertheless reveals some differences, which can lead to different solutions:

- the hinge loss is equal to zero as soon as $yh(\mathbf{x}) \geq 1$, so the SVM solution only depends on the support vectors, whereas the logistic loss never vanishes, so every training example (even a well-classified one far from the separator) has an influence on the logistic regression solution;
- both losses grow *linearly* for badly misclassified points, so they have a comparable sensitivity to outliers; this is not the case of the exponential loss used in boosting, which penalizes errors exponentially and is thus much more sensitive to outliers and label noise;
- logistic regression directly provides an estimate of the probability $Pr(y = 1 \mid \mathbf{x})$, whereas the SVM only provides a score whose sign gives the class.

Compared to linear regression for real-valued outputs, there is no analytical solution here. However, the problem is convex, so we can use a gradient-based algorithm to find a solution.

Pitfall: separable data in logistic regression Convexity does not guarantee that a minimizer exists. If the training data are linearly separable by $(\mathbf{w}, b)$, *i.e.* $(2y_i - 1)(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) > 0$ for all i , then replacing $(\mathbf{w}, b)$ by $(t\mathbf{w}, tb)$ and letting $t \rightarrow +\infty$ makes every predicted probability tend to the true label, and the negative log-likelihood tends to 0 without ever reaching it: the maximum likelihood estimate does not exist [Albert and Anderson \(1984\)](#). In practice, the norm of the parameters keeps growing along the iterations, the estimated probabilities become (over-confident) 0's and 1's, and the software may report convergence warnings. Adding a regularization term $\lambda \|\mathbf{w}\|_2^2$, as described below, prevents the weights from diverging: the regularized objective is strictly convex in $\mathbf{w}$ and tends to $+\infty$ when $\|\mathbf{w}\|_2 \rightarrow \infty$, so that its minimizer exists.

Learning procedure To simplify the notation and avoid handling the offset b separately, we append a constant coordinate equal to 1 to each input and include b in the vector of parameters, *i.e.* we write $\mathbf{x} \leftarrow (1, \mathbf{x}) \in \mathbb{R}^{d+1}$ and $\mathbf{w} \leftarrow (b, \mathbf{w}) \in \mathbb{R}^{d+1}$, so that $\langle \mathbf{w}, \mathbf{x} \rangle + b$ becomes $\langle \mathbf{w}, \mathbf{x} \rangle$ and $g(\mathbf{w}, \mathbf{x}) = 1/(1 + \exp(-\langle \mathbf{w}, \mathbf{x} \rangle))$. We consider the negative log-likelihood $\ell(\mathbf{w}, S)$ without the factor $\frac{1}{m}$. Using the fact that the logistic function $\sigma(t) = 1/(1 + \exp(-t))$ satisfies $\sigma'(t) = \sigma(t)(1 - \sigma(t))$, we have $\nabla_{\mathbf{w}} \ln g(\mathbf{w}, \mathbf{x}) = (1 - g(\mathbf{w}, \mathbf{x}))\mathbf{x}$ and $\nabla_{\mathbf{w}} \ln(1 - g(\mathbf{w}, \mathbf{x})) = -g(\mathbf{w}, \mathbf{x})\mathbf{x}$. We can thus compute the gradient of ℓ with respect to $\mathbf{w}$:

$$\begin{aligned} \nabla \ell(\mathbf{w}, S) &= \begin{bmatrix} \frac{\partial \ell}{\partial w_1}(\mathbf{w}, S) \\ \frac{\partial \ell}{\partial w_2}(\mathbf{w}, S) \\ \vdots \\ \frac{\partial \ell}{\partial w_{d+1}}(\mathbf{w}, S) \end{bmatrix} \\ &\quad \downarrow \text{using the derivatives of } \ln g \text{ and } \ln(1 - g) \\ &= - \sum_{i=1}^m [y_i(1 - g(\mathbf{w}, \mathbf{x}_i))\mathbf{x}_i - (1 - y_i)g(\mathbf{w}, \mathbf{x}_i)\mathbf{x}_i] \\ &\quad \downarrow \text{simplifying} \\ &= - \sum_{i=1}^m \left(y_i - \frac{1}{1 + \exp(-\langle \mathbf{w}, \mathbf{x}_i \rangle)} \right) \mathbf{x}_i. \end{aligned}$$

Each example thus contributes proportionally to its “residual” $y_i - g(\mathbf{w}, \mathbf{x}_i)$, *i.e.* to the difference between its label and its predicted probability.

We can then apply the gradient descent algorithm, using the above expression of the gradient of the negative log-likelihood (which plays the role of our loss function in this case):

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla \ell(\mathbf{w}^{(k)}, S), \quad k = 0, 1, 2, \dots,$$

where $\eta > 0$ is the learning rate.

In practice, we often use the **Newton-Raphson** method to minimize the loss function, *i.e.* we replace the learning rate η by the inverse of the Hessian matrix of ℓ in the update. This Hessian matrix is given by

$$\nabla^2 \ell(\mathbf{w}, S) = \begin{bmatrix} \frac{\partial^2 \ell}{\partial w_1^2}(\mathbf{w}, S) & \cdots & \frac{\partial^2 \ell}{\partial w_1 \partial w_{d+1}}(\mathbf{w}, S) \\ \frac{\partial^2 \ell}{\partial w_2 \partial w_1}(\mathbf{w}, S) & \cdots & \frac{\partial^2 \ell}{\partial w_2 \partial w_{d+1}}(\mathbf{w}, S) \\ \vdots & & \vdots \\ \frac{\partial^2 \ell}{\partial w_{d+1} \partial w_1}(\mathbf{w}, S) & \cdots & \frac{\partial^2 \ell}{\partial w_{d+1}^2}(\mathbf{w}, S) \end{bmatrix},$$

and a direct computation, using again $\sigma' = \sigma(1 - \sigma)$, gives

$$\nabla^2 \ell(\mathbf{w}, S) = \sum_{i=1}^m g(\mathbf{w}, \mathbf{x}_i) (1 - g(\mathbf{w}, \mathbf{x}_i)) \mathbf{x}_i \mathbf{x}_i^T.$$

We can express the Hessian matrix in a more compact form:

$$\nabla^2 \ell(\mathbf{w}, S) = \mathbf{X}^T \mathbf{G} \mathbf{X},$$

where $\mathbf{X} \in \mathbb{R}^{m \times (d+1)}$ is the design matrix (*i.e.* the matrix whose rows are the examples $\mathbf{x}_i^T$) and $\mathbf{G} \in \mathbb{R}^{m \times m}$ is the diagonal matrix defined by

$$\mathbf{G} = \begin{bmatrix} g(\mathbf{w}, \mathbf{x}_1)(1 - g(\mathbf{w}, \mathbf{x}_1)) & 0 & \cdots & 0 \\ 0 & g(\mathbf{w}, \mathbf{x}_2)(1 - g(\mathbf{w}, \mathbf{x}_2)) & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & g(\mathbf{w}, \mathbf{x}_m)(1 - g(\mathbf{w}, \mathbf{x}_m)) \end{bmatrix}.$$

Note that, with the first expression, the Hessian matrix is a nonnegative linear combination of the rank-one positive semi-definite matrices $\mathbf{x}_i \mathbf{x}_i^T$, and is thus itself positive semi-definite: this proves that ℓ is convex. The Newton-Raphson algorithm then reads

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \left(\nabla^2 \ell(\mathbf{w}^{(k)}, S) \right)^{-1} \nabla \ell(\mathbf{w}^{(k)}, S),$$

which converges much faster (quadratically, near the solution) than the standard gradient descent algorithm with a constant step size, at the price of solving a linear system of size $d+1$ at each iteration. In the context of logistic regression, this algorithm is also known as *Iteratively Reweighted Least Squares* (IRLS).

Let us now say a few words about regularization, which is commonly used in both regression and SVM models¹³.

Regularization In order to avoid overfitting, a regularization term of the form $\lambda \|\mathbf{w}\|$ (or $\lambda \|\mathbf{w}\|^2$), where $\lambda > 0$ is a hyperparameter, is usually added to the objective, in regression as well as in classification tasks. The optimization problem of logistic regression can then be rewritten as

$$\min_{(\mathbf{w}, b) \in \mathbb{R}^{d+1}} -\frac{1}{m} \sum_{i=1}^m [y_i \ln(g(\mathbf{w}, b, \mathbf{x}_i)) + (1 - y_i) \ln(1 - g(\mathbf{w}, b, \mathbf{x}_i))] + \lambda \|\mathbf{w}\|^2.$$

For linear regression, it can be written as

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^{d+1}} \|\mathbf{y} - h(\boldsymbol{\theta}, \mathbf{X})\|_2^2 + \lambda \|\boldsymbol{\theta}\|^2 = \min_{\boldsymbol{\theta} \in \mathbb{R}^{d+1}} \sum_{i=1}^m (y_i - h(\boldsymbol{\theta}, \mathbf{x}_i))^2 + \lambda \|\boldsymbol{\theta}\|^2.$$

Note that the offset (b or θ_0) is usually not penalized.

In the two expressions above, the norm is not specified, but two norms are very often encountered: the L_1 norm $\|\cdot\|_1$ and the (squared) L_2 norm $\|\cdot\|_2^2$. The choice has an important impact on the model, and especially on the parameters it learns (see [Hastie et al. \(2009\)](#) for a detailed discussion):

- The L_1 norm is notably used to induce sparsity in the learned hypothesis, *i.e.* when we want the learned hypothesis to depend on as few parameters as possible. When we use this regularization term in linear regression, we talk about *Lasso regression*. This type of regularization is particularly useful for high-dimensional models, with many variables, as is the case in genetics: it highlights the most

¹³If you look at the soft margin SVM problem, the data-fitting term is the mean value of the slack variables (*i.e.* the average hinge loss), and the L_2 regularization term is related to the inverse of the margin of the model.

important variables for the prediction task. However, it has an important drawback: it is not differentiable at the points where some coordinates are equal to 0, and there is no closed-form solution in general, so specific optimization algorithms (*e.g.* subgradient or proximal methods) are required.

- The squared L_2 norm is used to prevent some parameters from taking too large values compared to the other parameters of the model. In this case, we talk about *Ridge regression*. The objective remains differentiable and, for linear regression, a closed-form solution is still available: $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}$, where the matrix to invert is always invertible when $\lambda > 0$ (for simplicity, this formula also penalizes the offset θ_0).

Ridge regression was introduced by [Hoerl and Kennard \(1970\)](#) to stabilize least squares when the features are strongly correlated, and the Lasso by [Tibshirani \(1996\)](#).

In a nutshell: why does the L_1 norm produce sparse solutions? By Lagrangian duality (Section 4.5), minimizing the empirical risk plus $\lambda \|\boldsymbol{\theta}\|$ amounts to minimizing the empirical risk over a ball $\{\boldsymbol{\theta} : \|\boldsymbol{\theta}\| \leq t\}$, for some radius t depending on λ . The solution is the first point where the level sets of the risk (ellipses, for least squares) touch this ball, as shown in Figure 67. The L_2 ball is round, so the contact point generally has all its coordinates nonzero: Ridge *shrinks* all the coefficients. The L_1 ball is a diamond (a cross-polytope in dimension d) whose corners lie on the axes, and the ellipses very often touch it first at a corner, where some coordinates are exactly zero: the Lasso *selects* variables.

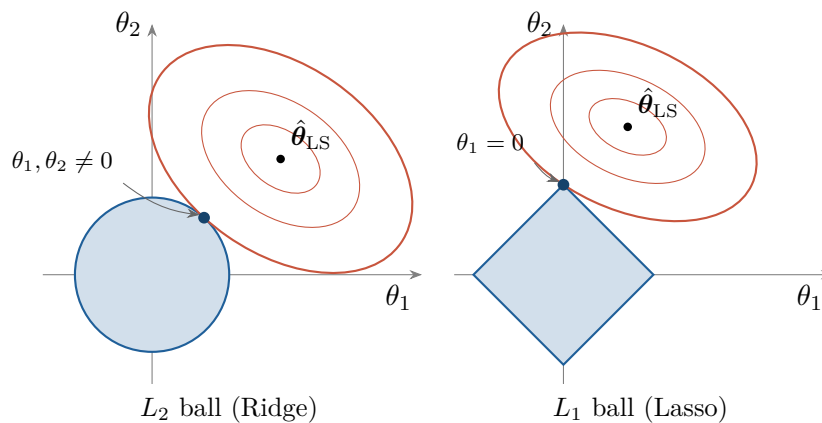


Figure 67: Geometry of Ridge (left) and Lasso (right) regularization in dimension 2. The red ellipses are level sets of the least squares objective, centered at the unregularized solution $\hat{\boldsymbol{\theta}}_{LS}$; the regularized solution (dark dot) is the point of the ball reached by the smallest level set. The corners of the L_1 ball lie on the axes, which produces exact zeros.

Before moving on to another algorithm, we extend *binary logistic regression* to multi-class classification.

8.5 Multinomial Logistic Regression

The previous classification algorithm was presented in the context of binary classification, and we may wonder how to adapt this procedure to multi-class classification

tasks, *i.e.* how to define a *multinomial logistic regression* Engel (1988). To this aim, let us assume that the set of labels is now $\{1, 2, \dots, q\}$, where $q > 2$. As before, we include the offset in the vector of parameters, $\mathbf{w} = (w_0, w_1, \dots, w_d)$, and we write $\mathbf{x} = (1, x_1, \dots, x_d)$.

In the binary case, we have built a linear model to estimate the logarithm of the ratio between the probability of belonging to class 1 and the probability of belonging to class 0 (also called a *logit*). We now have to do something similar with more than two classes: we select one class as a reference class, say class q , and we build the following models:

$$\begin{aligned} \ln \left(\frac{Pr(y = 1 | \mathbf{x})}{Pr(y = q | \mathbf{x})} \right) &= \langle \mathbf{w}^{(1)}, \mathbf{x} \rangle, \\ \ln \left(\frac{Pr(y = 2 | \mathbf{x})}{Pr(y = q | \mathbf{x})} \right) &= \langle \mathbf{w}^{(2)}, \mathbf{x} \rangle, \\ &\vdots \\ \ln \left(\frac{Pr(y = q-1 | \mathbf{x})}{Pr(y = q | \mathbf{x})} \right) &= \langle \mathbf{w}^{(q-1)}, \mathbf{x} \rangle. \end{aligned}$$

From this set of equations, we can recover the value of $Pr(y = k | \mathbf{x})$ for all $k = 1, \dots, q$. Indeed, if we take the exponential of each equation, sum all of them, and use the fact that $\sum_{k=1}^{q-1} Pr(y = k | \mathbf{x}) = 1 - Pr(y = q | \mathbf{x})$, we get

$$\frac{1 - Pr(y = q | \mathbf{x})}{Pr(y = q | \mathbf{x})} = \sum_{k=1}^{q-1} \exp \left(\langle \mathbf{w}^{(k)}, \mathbf{x} \rangle \right),$$

and thus

$$Pr(y = q | \mathbf{x}) = \frac{1}{1 + \sum_{k=1}^{q-1} \exp \left(\langle \mathbf{w}^{(k)}, \mathbf{x} \rangle \right)}.$$

Using this equality, we immediately obtain, for all $k \in \llbracket 1, q-1 \rrbracket$,

$$Pr(y = k | \mathbf{x}) = \frac{\exp \left(\langle \mathbf{w}^{(k)}, \mathbf{x} \rangle \right)}{1 + \sum_{l=1}^{q-1} \exp \left(\langle \mathbf{w}^{(l)}, \mathbf{x} \rangle \right)}.$$

Let us make a few remarks about this model:

- the probabilities defined above sum to 1;
- the number of parameters to learn is equal to the number of parameters of a single binary logistic regression times the number of classes minus one, *i.e.* $(q-1) \times (d+1)$;
- the way the model is built depends on the choice of the reference class: if we change it, the parameters have no reason to remain the same (although the modeled probabilities can be the same).

A simple way to learn this model is to fit $q-1$ binary logistic regressions, the k -th one opposing class k to the reference class q (on the examples of these two classes only), which gives a simple approximation of the joint estimate; the parameters can also be estimated jointly, by maximizing the multinomial likelihood described below. At the

prediction step, we compute all the probabilities $Pr(y = k \mid \mathbf{x})$, $k = 1, \dots, q$, and a new example $\mathbf{x}$ is assigned to the class with the highest probability, *i.e.*

$$\hat{y} = \arg \max_{k \in \llbracket 1, q \rrbracket} Pr(y = k \mid \mathbf{x}).$$

The above procedure has the advantage of reusing the tools presented in the binary setting. However, it requires learning $q - 1$ models which depend on a reference class. To address this last issue, and make our model independent of the reference class q , we can consider a parameter vector $\mathbf{w}^{(k)}$ for every class and define the probabilities by

$$Pr(y = k \mid \mathbf{x}) = \frac{\exp(\langle \mathbf{w}^{(k)}, \mathbf{x} \rangle)}{\sum_{l=1}^q \exp(\langle \mathbf{w}^{(l)}, \mathbf{x} \rangle)}, \quad \forall k = 1, \dots, q.$$

This function is known as the **softmax** function (it can be seen as a generalization of the logistic function), and the probabilities still sum to one. Note that adding the same vector to all the $\mathbf{w}^{(k)}$ does not change these probabilities: the parameters are thus only defined up to such a shift. Setting $\mathbf{w}^{(q)} = \mathbf{0}$ gives back the model with reference class q , while adding a regularization term is another common way to remove this indeterminacy. For $q = 2$, the softmax reduces to the logistic function applied to $\langle \mathbf{w}^{(1)} - \mathbf{w}^{(2)}, \mathbf{x} \rangle$.

It remains to explain the word *multinomial* in the name of the model. In the binary case, we have assumed that the label is drawn according to a Bernoulli distribution (*i.e.* a binomial distribution with $n = 1$). With q classes, the label of an example $\mathbf{x}$ now follows a categorical distribution (*i.e.* a multinomial distribution with $n = 1$) whose parameters are the q probabilities $Pr(y = k \mid \mathbf{x})$, $k = 1, \dots, q$. Denoting by $\mathbf{W} = (\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(q)})$ the matrix of all the parameters, the likelihood $\mathfrak{L}$ of our data is therefore based on the *multinomial distribution* and defined by

$$\begin{aligned} \mathfrak{L}(\mathbf{W}, S) &= \prod_{i=1}^m \prod_{k=1}^q Pr(y = k \mid \mathbf{x}_i)^{\mathbb{1}_{\{y_i=k\}}} \\ &= \prod_{i=1}^m \prod_{k=1}^q \left(\frac{\exp(\langle \mathbf{w}^{(k)}, \mathbf{x}_i \rangle)}{\sum_{l=1}^q \exp(\langle \mathbf{w}^{(l)}, \mathbf{x}_i \rangle)} \right)^{\mathbb{1}_{\{y_i=k\}}}. \end{aligned}$$

The negative log-likelihood ℓ , also called the *cross-entropy loss*, can then be written as

$$\ell(\mathbf{W}, S) = - \sum_{i=1}^m \sum_{k=1}^q \mathbb{1}_{\{y_i=k\}} \ln \left(\frac{\exp(\langle \mathbf{w}^{(k)}, \mathbf{x}_i \rangle)}{\sum_{l=1}^q \exp(\langle \mathbf{w}^{(l)}, \mathbf{x}_i \rangle)} \right). \quad (8.18)$$

The estimation of the parameters works exactly as in the binary case: there is no analytical solution to the minimization of (8.18), and we usually use a gradient descent or a Newton-Raphson algorithm. It is thus necessary to compute the gradient (and the Hessian matrix) of ℓ with respect to all the parameters. Denoting by $p_k(\mathbf{x}_i)$ the softmax probability of class k for the example $\mathbf{x}_i$, the gradient with respect to $\mathbf{w}^{(k)}$ takes a form very similar to the binary case:

$$\nabla_{\mathbf{w}^{(k)}} \ell(\mathbf{W}, S) = - \sum_{i=1}^m \left(\mathbb{1}_{\{y_i=k\}} - p_k(\mathbf{x}_i) \right) \mathbf{x}_i, \quad \forall k = 1, \dots, q.$$

In the next section, we focus on a nonparametric counterpart of regression, called *kernel regression*.

8.6 Kernel Regression

The linear and logistic regression models presented in the previous sections are both **parametric models**, *i.e.* we need to learn (or estimate) a fixed number of parameters, which determine the shape of the model, in order to make predictions.

There also exist **nonparametric models**, which do not rely on a fixed set of parameters. The prediction for a new instance is then directly based on the training instances and, more precisely, on those which are the most similar to it. The k -NN algorithm is an example of such a model.

Let us consider the problem of estimating the value of y given the value of $\mathbf{x}$; for the illustration, $\mathbf{x}$ is a real number. The data, together with the linear regression model learned from them, are represented in Figure 68.

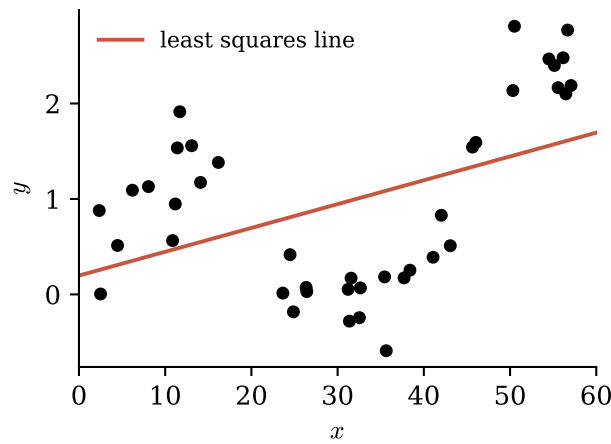


Figure 68: Representation of the data and of the linear model learned to fit them. This model is not appropriate, and it seems that we should use a nonlinear one. `[code: make_figures_ch07.py: fig_linreg]`

Using a linear model to estimate the values of y does not seem appropriate here. How would a nonparametric model perform in this case?

Let us introduce a first simple model. To estimate the value $y(\mathbf{x})$ associated with $\mathbf{x}$, we simply compute the mean of the values $y_i = y(\mathbf{x}_i)$ of the neighbors of $\mathbf{x}$, *i.e.* of the training examples lying at a distance at most λ from $\mathbf{x}$. The estimator h is thus given by

$$h(\mathbf{x}) = \frac{\sum_{i=1}^m K_{\lambda}(\mathbf{x}, \mathbf{x}_i) y_i}{\sum_{i=1}^m K_{\lambda}(\mathbf{x}, \mathbf{x}_i)},$$

where, in this case, $K_{\lambda}(\mathbf{x}, \mathbf{x}_i) = 1$ if the distance $\|\mathbf{x} - \mathbf{x}_i\|$ between $\mathbf{x}$ and $\mathbf{x}_i$ is lower than or equal to λ , and 0 otherwise. This estimator is a local average, but it has two drawbacks: it is discontinuous (a training point suddenly enters or leaves the neighborhood when $\mathbf{x}$ moves), and it is not defined when no training point lies within distance λ of $\mathbf{x}$. A natural idea is to replace this “all-or-nothing” weight by a weight which decreases smoothly with the distance. This leads to the following definition, due independently to [Nadaraya \(1964\)](#) and [Watson \(1964\)](#).

Definition 8.3: Kernel regression (Nadaraya-Watson estimator)

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m \subset \mathbb{R}^d \times \mathbb{R}$ be a sample of size m , let $\lambda > 0$ be a bandwidth, and let $K_\lambda(\mathbf{x}, \mathbf{x}') = D\left(\frac{\|\mathbf{x} - \mathbf{x}'\|}{\lambda}\right)$, where $D : \mathbb{R}_+ \rightarrow \mathbb{R}_+$ is a nonincreasing function. The kernel regression estimator is defined, for all $\mathbf{x}$ such that the denominator is positive, by

$$h(\mathbf{x}) = \frac{\sum_{i=1}^m K_\lambda(\mathbf{x}, \mathbf{x}_i) y_i}{\sum_{i=1}^m K_\lambda(\mathbf{x}, \mathbf{x}_i)} = \sum_{i=1}^m \omega_i(\mathbf{x}) y_i, \quad \text{with} \quad \omega_i(\mathbf{x}) = \frac{K_\lambda(\mathbf{x}, \mathbf{x}_i)}{\sum_{j=1}^m K_\lambda(\mathbf{x}, \mathbf{x}_j)}.$$

The prediction is thus a weighted average of the training outputs, whose weights $\omega_i(\mathbf{x})$ are nonnegative, sum to one, and are larger for the training points close to $\mathbf{x}$. Equivalently, $h(\mathbf{x})$ is the constant c which minimizes the locally weighted squared error $\sum_{i=1}^m K_\lambda(\mathbf{x}, \mathbf{x}_i)(y_i - c)^2$: kernel regression fits a constant model locally, around each query point (see [Hastie et al. \(2009\)](#) for extensions such as local linear regression). Common choices for D are the “box” function $D(t) = \mathbb{1}_{\{t \leq 1\}}$, which gives back the first estimator, and the Gaussian function $D(t) = \exp(-t^2/2)$, which gives $K_\lambda(\mathbf{x}, \mathbf{x}_i) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2\lambda^2}\right)$ (multiplicative constants do not matter since they cancel out in the ratio).

The bandwidth λ plays the same role as k in the k -NN algorithm and controls the bias-variance trade-off:

- when λ is small, only the very close points are taken into account: the estimator follows the data closely and is very wiggly (low bias, high variance, risk of overfitting);
- when λ is large, all the points receive almost the same weight: the estimator becomes almost constant, close to the mean of the y_i 's (high bias, low variance, risk of underfitting).

As usual, λ is selected by cross-validation. Note that, like the k -NN algorithm, kernel regression requires storing the whole training set and computing m distances for each prediction.

Example 8.7. Let us consider the four training points $(1, 1)$, $(2, 3)$, $(3, 2)$ and $(4, 6)$, and let us predict the output at $x = 2.5$.

- With the box kernel and $\lambda = 1$, only the points $x_2 = 2$ and $x_3 = 3$ are within distance 1 of x , so $h(2.5) = \frac{3+2}{2} = 2.5$.
- With the box kernel and $\lambda = 2$, all four points are taken into account, and $h(2.5) = \frac{1+3+2+6}{4} = 3$, the global mean.
- With the Gaussian kernel and $\lambda = 1$, the weights are $\exp(-1.5^2/2) \simeq 0.32$ for x_1 and x_4 , and $\exp(-0.5^2/2) \simeq 0.88$ for x_2 and x_3 , so

$$h(2.5) \simeq \frac{0.32 \times 1 + 0.88 \times 3 + 0.88 \times 2 + 0.32 \times 6}{2 \times 0.32 + 2 \times 0.88} \simeq 2.77.$$

All the points contribute to the Gaussian estimate, but the closest ones contribute more.

Figure 69 shows the estimators obtained with these two kernels on the data of Figure 68: both follow the nonlinear trend of the data, and the Gaussian kernel gives a much smoother curve than the box kernel. Its right panel illustrates the role of the bandwidth: a very small λ gives an irregular estimator that follows the noise, while a very large λ gives an almost constant one.

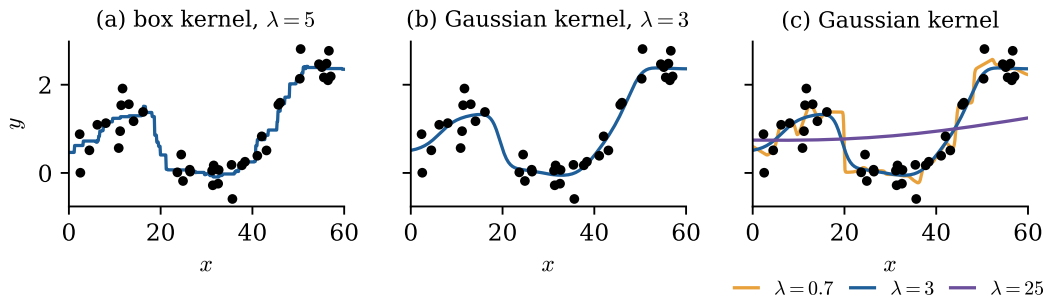


Figure 69: Kernel regression (Nadaraya-Watson) estimators on the data of Figure 68: (a) box kernel $K_\lambda(\mathbf{x}, \mathbf{x}_i) = \mathbb{1}_{\{\|\mathbf{x} - \mathbf{x}_i\| \leq \lambda\}}$ with $\lambda = 5$; (b) Gaussian kernel $K_\lambda(\mathbf{x}, \mathbf{x}_i) = \exp(-\|\mathbf{x} - \mathbf{x}_i\|^2 / (2\lambda^2))$ with $\lambda = 3$; (c) Gaussian kernel with a too small ($\lambda = 0.7$, overfitting), a reasonable ($\lambda = 3$) and a too large ($\lambda = 25$, underfitting) bandwidth. [code: make_figures_ch07.py: fig_kernel_regression]

The next algorithm we present can be used for both regression and classification tasks. It can return the probability of belonging to a given class, like logistic regression, or directly assign a label, like the SVM or the k -NN algorithm.

8.7 Decision Trees (classification and regression)

Two classical families of decision tree algorithms coexist: the Classification and Regression Trees (CART) algorithm introduced by Breiman et al. (1984), which is the version most widely used today, and the ID3 algorithm (and its successors) introduced by Quinlan (1986).

A decision tree consists of a series of rules which are successively applied to the dataset in order to split the data into two or more groups. Here, we only focus on binary decision trees, *i.e.* trees in which each decision rule splits the dataset into exactly two subsets.

The nature of the tree depends on the output space $\mathcal{Y}$:

- when $\mathcal{Y} \subset \mathbb{R}$, we talk about a regression tree;
- when $\mathcal{Y}$ is a finite set of classes, *e.g.* $\mathcal{Y} = \{-1, 1\}$, we talk about a classification tree.

An example of classification tree is provided in Figure 70 on a toy dataset, in which we aim to separate two classes (male and female) using two descriptors (age and height) and a set of rules. The initial dataset is composed of 4 females and 3 males. The first decision rule, $Age \leq 25$, divides the initial dataset into two groups: one composed of two females, and the other one of 3 males and 2 females. The group on the left is *pure*, since it only contains females, so we no longer need to split it. We then apply a second decision rule, $Height \leq 170$, which separates the remaining males and females.

To predict the label of a new example, we simply follow the rules from the root to a leaf, and predict the majority class of that leaf (the proportion of each class in the leaf can also be used as an estimate of the class probabilities).

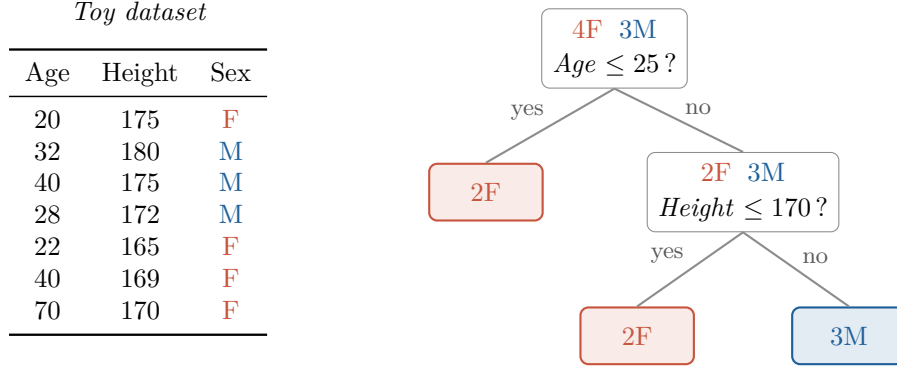


Figure 70: An example of classification tree learned on the toy dataset on the left (F: female, M: male). Each internal node shows its class counts and its decision rule; the three leaves are pure.

Such an algorithm is able to separate complex (not linearly separable) datasets by using a large number of decision rules. To choose the decision rules, we define a criterion to optimize.

For this purpose, we need two tools: a *metric* which evaluates the quality (or impurity) of a node, and a measure of the improvement obtained after a split, called the *gain* [Safavian and Landgrebe \(1991\)](#); [Rokach and Maimon \(2005\)](#).

The metric depends on the type of tree we are dealing with. A list of such metrics is provided by [Rokach and Maimon \(2005\)](#), among which:

- the *variance*, used for regression trees:

$$\frac{1}{m_N} \sum_{i=1}^{m_N} (y_i - \bar{y})^2,$$

where m_N denotes the number of examples in the node N and $\bar{y}$ the average value of the y_i 's in this node;

- the *entropy*, a notion also used in physics to quantify the disorder of a system, defined by $Ent_N = -\sum_{j=1}^L p_j \log_2(p_j)$ (with the convention $0 \log_2 0 = 0$), where p_j is the proportion of examples of class j in the node N and L is the number of classes;
- the *Gini impurity*, used for classification trees. It measures the impurity of a node from the proportions of the different classes present in the node. For instance, in binary classification¹⁴, the Gini impurity G_N of a node N is defined by

$$G_N = \sum_{j \in \{-1, 1\}} p_j(1 - p_j) = 2p_1(1 - p_1), \quad (8.19)$$

where p_j denotes the proportion of examples of class j in the node.

¹⁴The definition can be extended to L classes: the Gini impurity is then defined by $G_N = \sum_{j=1}^L p_j(1 - p_j) = 1 - \sum_{j=1}^L p_j^2$, where p_j is the proportion of examples of class j in the node N .

In the binary setting, the Gini impurity is a real number which belongs to $[0, 0.5]$. A value of 0 means that the node is pure, *i.e.* it only contains examples from one class. A value of 0.5 means that the node contains the same number of examples from both classes. Similarly, the binary entropy belongs to $[0, 1]$, with the same interpretation of its extreme values.

Let us now illustrate this notion. In the previous example (Figure 70), the Gini impurity of the root is $G_{\text{root}} = 2 \times \frac{4}{7} \left(1 - \frac{4}{7}\right) = \frac{24}{49} \simeq 0.49$, while it is equal to 0 and $\frac{12}{25}$ respectively in the two nodes obtained after the first split. We have already said that the node on the left is *pure*, because it only contains examples from one class; its Gini impurity therefore cannot be improved.

The next step consists in choosing the optimal rule to split a node into two child nodes. This rule is chosen so as to decrease the impurity as much as possible. For this purpose, we define the *Gini gain* Γ of a split of a node N (here the root) into two nodes N_L and N_R as follows:

$$\Gamma = G_{\text{root}} - \left(\frac{|N_L|}{|N_L| + |N_R|} G_{N_L} + \frac{|N_R|}{|N_L| + |N_R|} G_{N_R} \right),$$

where $|N_L|$ and $|N_R|$ denote the numbers of examples in the left and right nodes, and G_{N_L} and G_{N_R} their Gini impurities. In our example, the first split has a gain $\Gamma = \frac{24}{49} - \frac{5}{7} \times \frac{12}{25} = \frac{36}{245} \simeq 0.15$ (with the entropy instead of the Gini impurity, the gain would be $0.985 - \frac{5}{7} \times 0.971 \simeq 0.29$).

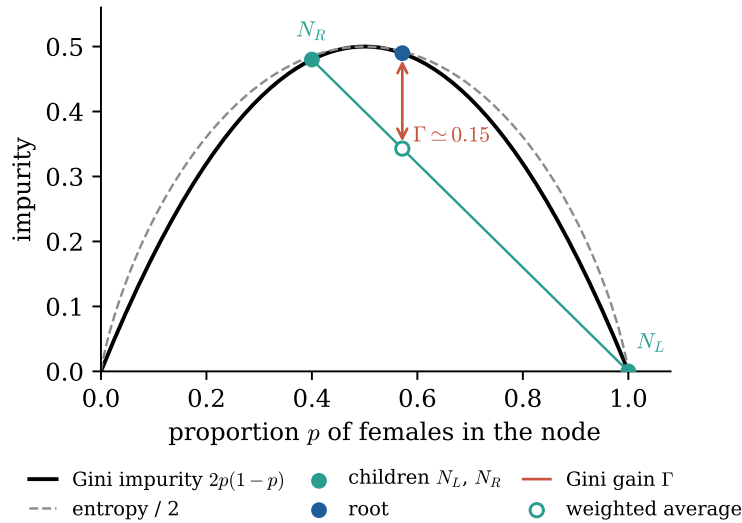


Figure 71: Illustration of the Gini gain using the first split of the example presented in Figure 70, as a function of the proportion p of females in a node. The root ($p = 4/7$) is split into N_L ($p = 1$) and N_R ($p = 2/5$); the weighted average of the impurities of the children lies on the green chord, and the red arrow represents the Gini gain Γ . The (halved) entropy has a very similar shape.

[code: make_figures_ch07.py: fig_gini]

Figure 71 illustrates the use of the Gini impurity as a metric to build our decision tree on this example. The red arrow represents the Gini gain Γ . This figure also shows that the Gini impurity is a concave function of the proportion p_1 . Since the proportion of a class in the parent node is the weighted average of its proportions in the two child

nodes, this concavity ensures, by Jensen's inequality [Jensen \(1906\)](#) (Proposition 1.3 in Section 1), that the gain is always nonnegative: a split never increases the (weighted) impurity. Note, however, that a split does not necessarily decrease the classification error of the tree, which is precisely why a concave impurity measure is used instead of the error rate. At each step, we choose the feature and the corresponding threshold which maximize the gain Γ . The decision rule is then applied, and the nodes are recursively split until pure leaves are obtained.

In practice, it is always possible to obtain such perfect leaves (unless two identical inputs have different labels). However, such trees tend to overfit and to generalize poorly. To overcome this issue, we usually restrict the growth of the tree (pre-pruning) using hyperparameters such as:

- the maximal *size/depth* of the tree;
- the size of a node: the minimum number of examples required in a node to split it;
- the size of a leaf: the minimum number of examples in each of the two leaves after a split;
- a threshold on the gain: the minimum value of the gain required to make a new split.

Another strategy (post-pruning) consists in growing a full tree and then removing the branches which do not improve the performance enough, as done in the cost-complexity pruning of CART [Breiman et al. \(1984\)](#).

Geometrically, each decision rule of the form $x_j \leq s$ cuts a region of the input space into two by a hyperplane orthogonal to the j -th axis: a tree thus partitions the space into axis-aligned rectangles $R_1, \dots, R_L$, one per leaf, and predicts a constant (a class or a value) in each of them, as shown in Figure 72.

Pitfall: deep trees overfit and are unstable A fully grown tree reaches a zero training error (when no two identical inputs have different labels), but, as in the right panel of Figure 72, it does so by creating tiny leaves around noisy examples: its test error is much larger than its training error. Deep trees also have a high variance in another sense: a small change in the training set can change the first split, and hence the whole tree. Therefore, never select the depth of a tree on the training error; tune the pruning hyperparameters by cross-validation. This instability is also what ensemble methods such as bagging and random forests exploit by averaging many trees (Section 10).

The gain defined above can be used with any impurity measure: entropy, Gini impurity, variance, or any other measure you may want to define.

Before moving on to another topic, let us consider another example with the same dataset, where we now aim to predict the age of a person from his or her height, using a regression tree. We first compute the variance at the root of the tree; in our case, it is equal to $Var_{\text{root}} \simeq 245.71$. We now have to find the binary split for which the gain, defined by

$$\Gamma = Var_{\text{root}} - \left(\frac{|N_L|}{|N_L| + |N_R|} Var_{N_L} + \frac{|N_R|}{|N_L| + |N_R|} Var_{N_R} \right),$$

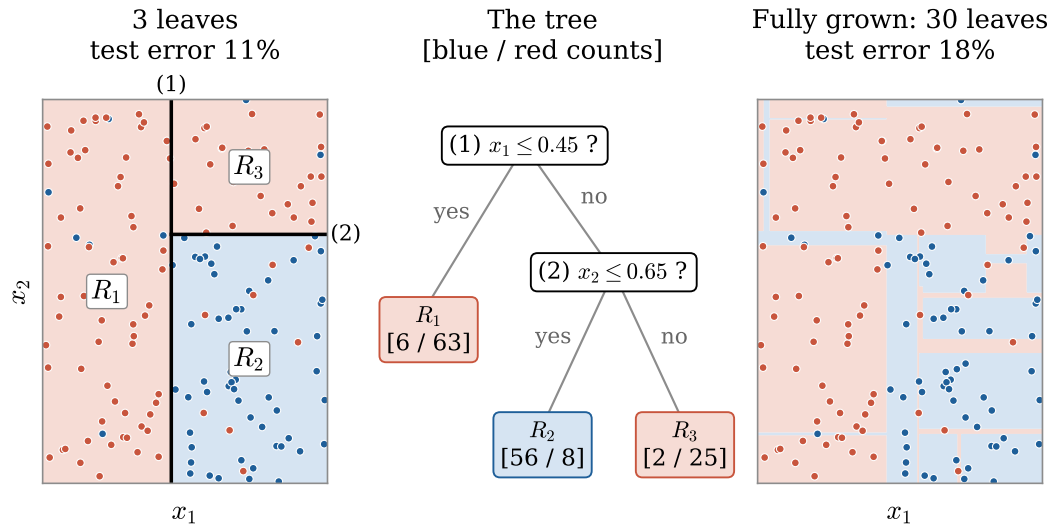


Figure 72: Left and middle: a small classification tree (three leaves) learned on 160 points of $[0, 1]^2$, 10% of whose labels have been flipped, and the corresponding partition of the input space into axis-aligned rectangles; (1) and (2) mark the two splits. Right: the fully grown tree has pure leaves, isolates the noisy points in small rectangles and has a larger test error. [code: `make_figures_ch07.py: fig_tree_partition`]

is the highest. For this purpose, we test the possible thresholds on the variable *height* and compute the associated values of Γ . The results of these computations are provided in Table 3 and show that, among the three tested splits, the best one is obtained with a threshold equal to 170. In a regression tree, the prediction associated with a leaf is the average output of the training examples it contains.

Threshold	$ N_L $	$ N_R $	Var_{N_L}	Var_{N_R}	Γ
169	2	5	81	297.6	10
170	3	4	392	52	48
172	4	3	342	67.56	21.33

Table 3: Values of the gain Γ , in terms of variance, for three different thresholds. An instance whose height is lower than or equal to the threshold is put in the left node, and in the right one otherwise.

Later in this document, we will see how decision trees can be combined to build stronger algorithms.

Summary Table 4 summarizes the main features of the algorithms presented in this section. There is no universally best algorithm: the choice depends on the size and dimension of the data, on the need for probabilities or for an interpretable model, and on the computational budget, and it should be validated experimentally (*e.g.* by cross-validation).

We conclude the presentation of the standard algorithms with Neural Networks, in the next section.

Algorithm	Task	Parametric	Output	Main hyperparameters
k -NN	classif./regression	no	label	k , distance
SVM	classification	yes*	label (score)	C , kernel
Linear regression	regression	yes	real value	regularization λ
Logistic regression	classification	yes	probability	regularization λ
Kernel regression	regression	no	real value	bandwidth λ , kernel
Decision tree	classif./regression	no	label/prob./value	depth, leaf size

* with a nonlinear kernel, the number of dual parameters grows with m .

Table 4: Summary of the supervised algorithms presented in this section.

To remember from Section 8

- The 0-1 loss is replaced by convex surrogates (hinge, logistic, exponential), written as functions of the margin $yh(\mathbf{x})$.
- k -NN is a nonparametric, memory-based rule; k controls the bias-variance trade-off, and the method requires scaled features and suffers from the curse of dimensionality.
- The SVM maximizes the margin $2/\|\mathbf{w}\|_2$; its soft margin version minimizes a regularized hinge loss. The dual only involves inner products, which the kernel trick replaces by a kernel K ; the solution only depends on the support vectors ($\alpha_i > 0$).
- Linear regression has a closed-form solution; logistic regression models the log-odds linearly, is fitted by maximum likelihood (gradient descent or Newton/IRLS), and extends to q classes with the softmax. L_2 regularization shrinks the coefficients, L_1 regularization sets some of them to zero.
- Kernel regression is a locally weighted average whose bandwidth λ plays the role of k .
- Decision trees recursively split the space into axis-aligned rectangles by maximizing an impurity gain (Gini, entropy, variance); they are interpretable but overfit when grown fully, hence pruning.
- In all cases, the hyperparameters (k , C , γ or σ , λ , depth) are selected by cross-validation.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

k -Nearest Neighbors

- Q8.1.** What is the training error of the 1-NN rule, and what does it tell us about its test error?
- Q8.2.** In binary classification, why is an odd value of k usually preferred? What happens with $k = 4$ when the vote is 2 against 2?
- Q8.3.** Data are uniform in $[0, 1]^5$. What is the edge length of a small hypercube around a query that contains 10% of the data? Comment.

Support Vector Machine

- Q8.4.** In the soft margin SVM, which training examples have $\alpha_i = 0$, and what is their influence on the classifier?
- Q8.5.** A linear SVM returns $\mathbf{w} = (3, 4)$. What is the width of its margin γ ?
- Q8.6.** True or false: increasing the hyperparameter C of the soft margin SVM can only widen the margin. Justify.
- Q8.7.** What happens to a Gaussian kernel SVM when σ becomes very small (*i.e.* γ very large)?

Multinomial Logistic Regression

- Q8.8.** Why does the Lasso produce exact zeros in the coefficients, whereas Ridge regression does not?
- Q8.9.** A logistic regression model gives the score $h(\mathbf{w}, b, \mathbf{x}) = \ln 3$ for an example $\mathbf{x}$. What is the estimated probability that $y = 1$? What are the odds?
- Q8.10.** How many parameters does a multinomial logistic regression with a reference class have for images of 28×28 pixels and $q = 10$ classes? And the softmax version with one vector per class?

Synthesis of the section

- Q8.11.** Why do we minimize a surrogate loss rather than the 0-1 loss itself?
- Q8.12.** Which algorithm of this section is insensitive to a rescaling of the features, and why?
- Q8.13.** What does the Nadaraya-Watson estimator with a Gaussian kernel become when $\lambda \rightarrow +\infty$? And when $\lambda \rightarrow 0$?
- Q8.14.** What is the Gini impurity of a node containing the same number of examples of 3 classes? Is it the maximal value for 3 classes?
- Q8.15.** A node contains 80 examples of class 1 and 20 of class -1 . It is split into a pure node (50 examples of class 1) and a node with 30 examples of class 1 and 20 of class -1 . What is the decrease of the misclassification error? Why do trees use the Gini impurity or the entropy instead?
- Q8.16.** Which algorithms of this section must keep (part of) the training set in memory to make predictions?

9 Neural Networks

Neural networks are the last family of models presented in this part. They can be used for classification and for regression, but also, as we will see at the end of the section, in unsupervised learning (dimensionality reduction, generative models). They are at the heart of what is now called *deep learning*, which has driven most of the recent progress in computer vision, speech recognition and natural language processing. A neural network is nothing more than a parametric function obtained by *composing* simple operations (affine maps and non-linear functions), whose parameters are learned by minimizing an empirical risk with a gradient-based method. The main ideas of this section are thus already familiar: a model, a loss and an optimization algorithm, see Sections 6 and 3. The reference textbook on the topic is the book of Goodfellow et al. (2016).

About this section

We start from the perceptron, the ancestor of all neural networks, and understand why it has to be stacked into several layers (Section 9.1). We then study the ingredients of a multilayer perceptron: activation functions (Section 9.2), its expressive power (Section 9.3) and the losses used to train it (Section 9.4). The core of the section is the backpropagation algorithm (Section 9.5) and the optimization and regularization techniques that make the training of deep networks possible (Sections 9.6 and 9.7). We finally present the main architectures: convolutional networks (Section 9.8), recurrent networks (Section 9.9) and a few others (Section 9.10), before giving some practical advice (Section 9.11).

9.1 From the perceptron to the multilayer perceptron

Foundations. Neural networks are (very freely) inspired by nature, and more precisely by neuroscience and models of synapses. The first traces of neural networks can be found in works from the middle of the 20th century: McCulloch and Pitts (1943) proposed the first mathematical model of a neuron, a unit that “fires” when the weighted sum of its inputs exceeds a threshold. A few years later, Rosenblatt (1958) proposed a way to *learn* the weights of such a unit from examples: this is the *perceptron*, represented in Fig. 73.

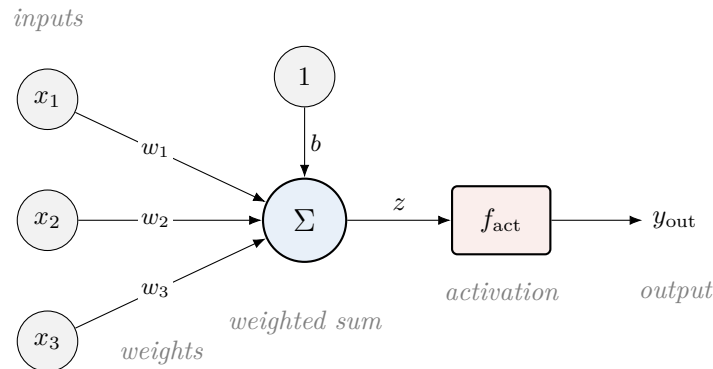


Figure 73: A perceptron with an input space of dimension 3. Each feature x_j is multiplied by a weight w_j and a bias b is added: $z = \sum_{j=1}^3 w_j x_j + b$. This weighted sum then goes through an activation function f_{act} to give the output $y_{\text{out}} = f_{\text{act}}(z)$.

The perceptron. A perceptron takes a vector $\mathbf{x} \in \mathbb{R}^d$ as input and depends on a parameter $(\mathbf{w}, b) \in \mathbb{R}^d \times \mathbb{R}$, where $\mathbf{w}$ is the vector of *weights* and b the *bias*. Historically, its output $h(\mathbf{x})$ is the sign of the inner product between $\mathbf{x}$ and $\mathbf{w}$, to which the bias b is added, *i.e.*

$$h(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) = \text{sign} \left(\sum_{j=1}^d w_j x_j + b \right).$$

This first model was thus designed for binary classification. It computes an affine function of the input (linear if $b = 0$), to which an *activation function* is applied to return the predicted class. With labels in $\{0, 1\}$, the prediction reads

$$h(\mathbf{x}) = \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b < 0, \\ 1 & \text{otherwise,} \end{cases}$$

and the activation function is the *Heaviside* (or step) function $H(z) = \mathbb{1}_{z \geq 0}$, represented in Fig. 77 (left). Note that this decision rule is exactly the one of the linear SVM seen in Section 8: the perceptron is a linear classifier, and only the way its parameters are learned differs. In the rest of this paragraph, we use labels in $\{-1, +1\}$ and $h(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$, which makes the formulas shorter.

The perceptron learning rule. The first algorithm used to learn the parameters $\mathbf{w}$ and b is inspired by the rule of the neuropsychologist Hebb (“neurons that fire together wire together”). Its principle is simple: the examples are visited one after the other, and the parameters are only modified when the current example is misclassified, *i.e.* when $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \leq 0$. In this case, we update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha y_i \mathbf{x}_i \quad \text{and} \quad b \leftarrow b + \alpha y_i,$$

where $\alpha > 0$ is the *learning rate*. Such an update moves the hyperplane in the right direction for the example $(\mathbf{x}_i, y_i)$, since the new score satisfies

$$y_i(\langle \mathbf{w} + \alpha y_i \mathbf{x}_i, \mathbf{x}_i \rangle + b + \alpha y_i) = y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) + \alpha(\|\mathbf{x}_i\|^2 + 1) > y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b).$$

The procedure is repeated until no example is misclassified, or stops after a given number of passes over the data (called *epochs*). The procedure is summarized in Algorithm 4. A classical result states that it stops after a finite number of updates when the data are linearly separable.

Theorem 9.1: Convergence of the perceptron (Novikoff, 1962)

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a training set with $\|\mathbf{x}_i\| \leq R$ for all i . Assume that the data are linearly separable with a margin $\gamma > 0$, *i.e.* there exists $\mathbf{w}^*$ with $\|\mathbf{w}^*\| = 1$ such that $y_i \langle \mathbf{w}^*, \mathbf{x}_i \rangle \geq \gamma$ for all i (no bias here, see the remark below). Then the perceptron (without bias, starting from $\mathbf{w} = \mathbf{0}$, with any $\alpha > 0$) makes at most R^2/γ^2 updates.

Proof. Let $\mathbf{w}_k$ be the weight vector after k updates, and let $(\mathbf{x}_i, y_i)$ be the example that triggers the $(k+1)$ -th update, so that $y_i \langle \mathbf{w}_k, \mathbf{x}_i \rangle \leq 0$. On the one hand, the projection on $\mathbf{w}^*$ increases at each update:

$$\langle \mathbf{w}_{k+1}, \mathbf{w}^* \rangle = \langle \mathbf{w}_k, \mathbf{w}^* \rangle + \alpha y_i \langle \mathbf{x}_i, \mathbf{w}^* \rangle \geq \langle \mathbf{w}_k, \mathbf{w}^* \rangle + \alpha \gamma, \quad \text{hence} \quad \langle \mathbf{w}_k, \mathbf{w}^* \rangle \geq k \alpha \gamma.$$

Algorithm 4: Perceptron learning rule (Rosenblatt, 1958)

Input: training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ with $y_i \in \{-1, +1\}$, learning rate $\alpha > 0$, maximal number of epochs T

Output: parameters $(\mathbf{w}, b)$ of the linear classifier $h(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$

```
1  $\mathbf{w} \leftarrow \mathbf{0}, b \leftarrow 0;$ 
2 for  $t = 1, \dots, T$  do
3   for  $i = 1, \dots, m$  do
4     if  $y_i (\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \leq 0$  then
5        $\mathbf{w} \leftarrow \mathbf{w} + \alpha y_i \mathbf{x}_i$  and  $b \leftarrow b + \alpha y_i;$ 
6   if no update has been made during the epoch then stop;
```

On the other hand, the norm does not increase too fast:

$$\|\mathbf{w}_{k+1}\|^2 = \|\mathbf{w}_k\|^2 + 2\underbrace{\alpha y_i \langle \mathbf{w}_k, \mathbf{x}_i \rangle}_{\leq 0} + \alpha^2 \|\mathbf{x}_i\|^2 \leq \|\mathbf{w}_k\|^2 + \alpha^2 R^2, \quad \text{hence} \quad \|\mathbf{w}_k\|^2 \leq k\alpha^2 R^2.$$

Combining both inequalities with the Cauchy–Schwarz inequality gives $k\alpha\gamma \leq \langle \mathbf{w}_k, \mathbf{w}^* \rangle \leq \|\mathbf{w}_k\| \leq \sqrt{k} \alpha R$, i.e. $k \leq R^2/\gamma^2$. $\square$

Remark The bias can be handled by adding a constant feature equal to 1 to each input: $\tilde{\mathbf{x}} = (\mathbf{x}, 1)$ and $\tilde{\mathbf{w}} = (\mathbf{w}, b)$. Note also that the number of updates does not depend on the dimension d nor on the number of examples m , but only on the geometry of the problem through the ratio R/γ . The margin γ is the same notion as the one maximized by the SVM. However, the perceptron returns *one* separating hyperplane, not the one with maximal margin, and it never stops (it cycles) when the data are not linearly separable.

The Widrow–Hoff rule. Another rule to learn the parameters of a perceptron, called the *Widrow–Hoff rule* (or *delta rule*), works in a similar manner but takes into account the error made on the current example:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (y_i - h(\mathbf{x}_i)) \mathbf{x}_i \quad \text{and} \quad b \leftarrow b + \alpha (y_i - h(\mathbf{x}_i)).$$

This rule can be used with the sign function, but it is usually combined with the hyperbolic tangent $\tanh$, which can be seen as a smooth version of the sign function (see Fig. 77), or with the sigmoid function, a smooth version of the Heaviside function. Smooth activation functions have a major advantage: the derivative of the Heaviside function is equal to 0 almost everywhere, so that it does not provide any information to a gradient descent algorithm, whereas smooth functions do. When h is the identity, the Widrow–Hoff rule is exactly a stochastic gradient step on the squared loss $\frac{1}{2}(y_i - \langle \mathbf{w}, \mathbf{x}_i \rangle - b)^2$, a point of view that we will generalize to any network in Section 9.5.

The limits of the perceptron: the XOR problem. The previous algorithms work well when the problem is linearly separable. This situation is rarely met in practice, and we need more flexible models to achieve good performance on non-linear problems.

For instance, the perceptron can perfectly classify the **OR** and **AND** datasets, since a linear model is enough to separate the two classes, but it cannot solve the **XOR** (exclusive or) problem. These three problems are illustrated in Fig. 74: no line can separate the two classes of the third dataset. This simple observation, formalized by [Minsky and Papert \(1969\)](#), contributed to a significant decline of research on neural networks in the 1970s. To solve the problem, we need a more complex model, one that *learns another representation of the data* in which the problem becomes linearly separable.

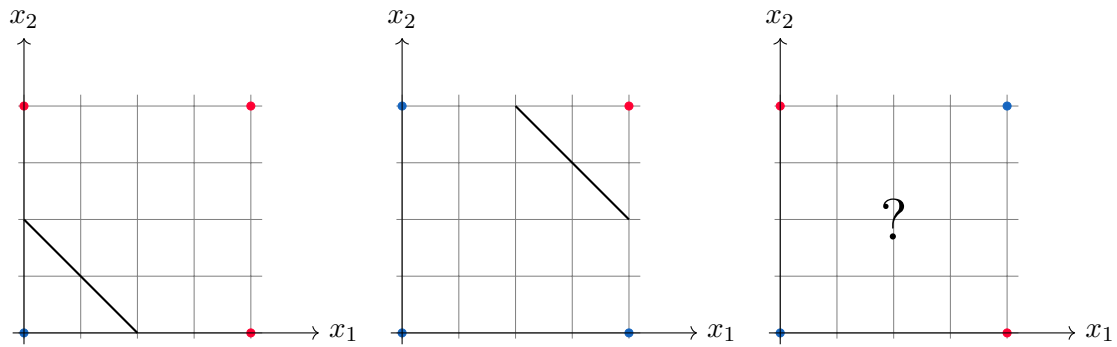
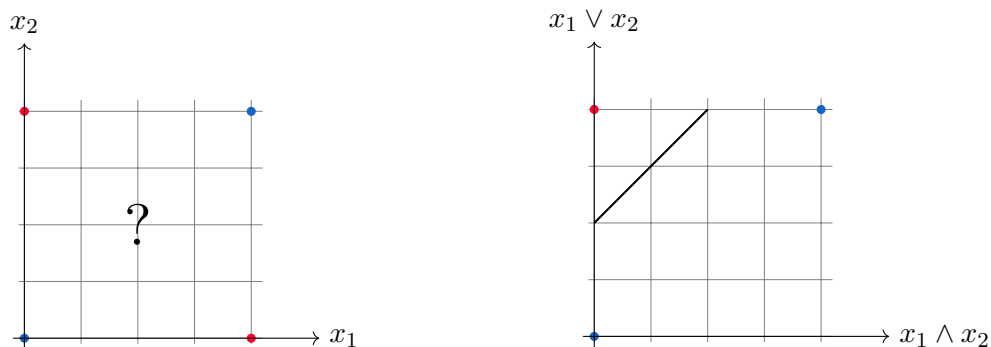


Figure 74: The **OR**, **AND** and **XOR** classification datasets (from left to right) in a two-dimensional space. A coordinate x_j equal to 0 means that the corresponding entry is *FALSE*; it is equal to 1 if the entry is *TRUE*. The color of a point gives its label, determined by the logical operator: **TRUE** or **FALSE**. A line separates the classes of the first two datasets, but no line can separate those of the **XOR** dataset.

To this end, we are again inspired by the architecture of the brain, where many neurons are connected to each other: this is what we call a *neural network*. Before describing the general architecture, let us go back to our example and see how the **XOR** problem can be solved. Consider the following transformation of the axes:

$$x_1 \leftarrow x_1 \wedge x_2 \quad \text{and} \quad x_2 \leftarrow x_1 \vee x_2,$$

which gives the following new representation of the data (the **XOR** dataset on the left, its new representation on the right):



With this transformation, the two red points are mapped to the same point of the new space, because exactly one of their two entries is *TRUE*. The blue points stay where they are, since their two entries are equal. In this new representation, the problem is linearly separable, and such a representation can be learned by a neural network, more precisely by a *multilayer perceptron*.

Function	Weights	Bias	Output
$x_1 \wedge x_2$ (AND)	$w_1 = w_2 = 1$	$b = -1.5$	$H(x_1 + x_2 - 1.5)$
$x_1 \vee x_2$ (OR)	$w_1 = w_2 = 1$	$b = -0.5$	$H(x_1 + x_2 - 0.5)$
$\bar{x}$ (NOT)	$w = -1$	$b = 0.5$	$H(-x + 0.5)$

Table 5: The elementary logical functions are perceptrons with the Heaviside activation $H(z) = \mathbf{1}_{z \geq 0}$ and inputs in $\{0, 1\}$.

To find a suitable architecture, let us first rewrite the **XOR** function. This function is *TRUE* if and only if exactly one of its inputs is *TRUE*. In other words,

$$\text{XOR}(x_1, x_2) = (x_1 \vee x_2) \wedge (\overline{x_1 \wedge x_2}).$$

It is now enough to translate this expression with several perceptrons using the Heaviside activation H : the elementary logical functions are all perceptrons, as shown in Table 5. Composing them gives the network of Fig. 75, whose hidden layer computes the new representation $(x_1 \vee x_2, x_1 \wedge x_2)$ and whose output unit computes “ $x_1 \vee x_2$ and not $x_1 \wedge x_2$ ”.

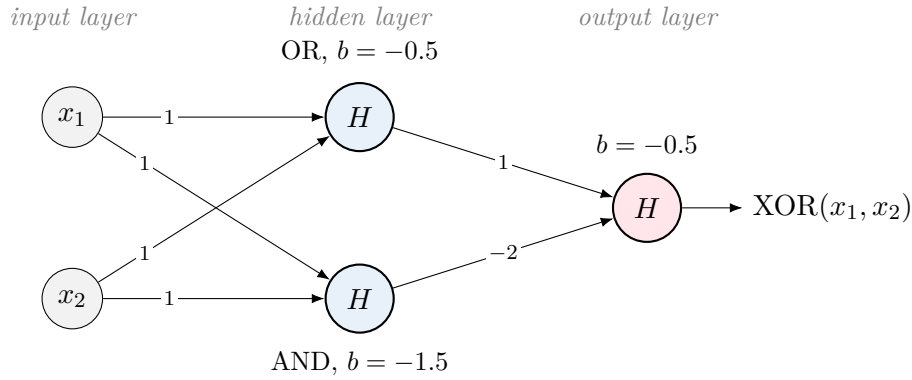


Figure 75: A multilayer perceptron with Heaviside activations solving the **XOR** problem. The hidden units compute $x_1 \vee x_2$ and $x_1 \wedge x_2$; the output unit computes $H((x_1 \vee x_2) - 2(x_1 \wedge x_2) - 0.5)$, which is equal to 1 if and only if exactly one input is equal to 1.

The multilayer perceptron. A multilayer perceptron (MLP) is represented in Fig. 76. The first layer is called the *input layer*. Its size is equal to the dimension of the input space, and we add another unit whose value is always equal to 1 and which represents the bias term. The intermediate layers are called *hidden layers*: their number and their sizes are chosen by the user according to the problem at hand. To each hidden layer, we can also associate a bias that is used to compute the values of the next layer. The last layer is called the *output layer*, and its size depends on the output space. For regression or binary classification, the output is of dimension 1. For multi-class classification, the output layer has as many units as there are classes in the dataset.

The network of Fig. 76 is said to be *fully connected*: every unit of a layer is connected to every unit of the next layer. It is possible to remove some of the connections,

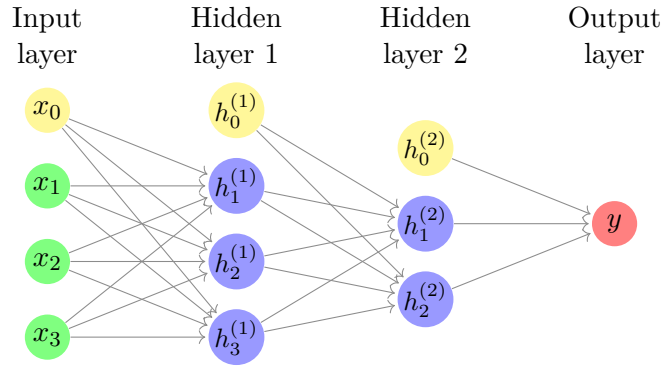


Figure 76: A multilayer perceptron. In this example, the input space is of dimension 3 and there are two hidden layers, of size 3 and 2 respectively. The output space is of dimension 1. The units labeled 1 represent the biases.

by setting the corresponding weights to 0, and we could also imagine connections between two non-consecutive layers (this is the idea of residual networks, see [Section 9.8](#)). Let us formalize the model.

Definition 9.1: Multilayer perceptron

Let $L \geq 1$ be the number of layers (the input layer is not counted) and $d_0 = d, d_1, \dots, d_L$ their sizes. A *multilayer perceptron* is a function $f_{\theta} : \mathbb{R}^d \rightarrow \mathbb{R}^{d_L}$ defined by $\mathbf{h}^{(0)} = \mathbf{x}$ and, for $\ell = 1, \dots, L$,

$$\mathbf{a}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \quad \text{and} \quad \mathbf{h}^{(\ell)} = \sigma_{\ell}(\mathbf{a}^{(\ell)}),$$

and $f_{\theta}(\mathbf{x}) = \mathbf{h}^{(L)}$, where

- $\mathbf{W}^{(\ell)} \in \mathbb{R}^{d_{\ell} \times d_{\ell-1}}$ is the weight matrix and $\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_{\ell}}$ the bias vector of layer ℓ ;
- $\sigma_{\ell} : \mathbb{R} \rightarrow \mathbb{R}$ is the activation function of layer ℓ , applied componentwise (σ_L can be the identity or the softmax function, see [Section 9.4](#));
- $\mathbf{a}^{(\ell)}$ is called the *pre-activation* and $\mathbf{h}^{(\ell)}$ the *activation* (or hidden representation) of layer ℓ .

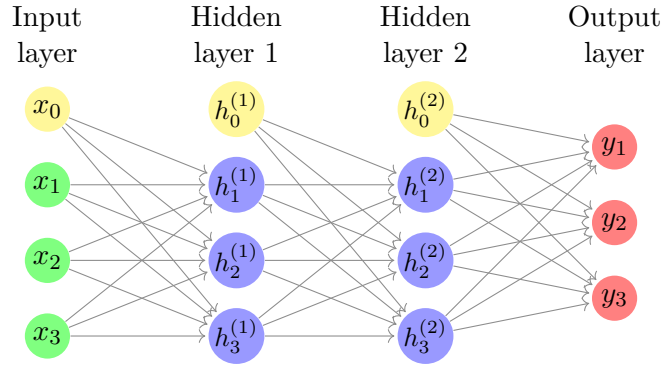
The parameter of the network is $\theta = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)})$. A network is called *deep* when it has several hidden layers.

In other words, a multilayer perceptron is the composition $f_{\theta} = \sigma_L \circ A_L \circ \sigma_{L-1} \circ A_{L-1} \circ \dots \circ \sigma_1 \circ A_1$, where $A_{\ell} : \mathbf{h} \mapsto \mathbf{W}^{(\ell)} \mathbf{h} + \mathbf{b}^{(\ell)}$ is an affine map. Each hidden layer computes a new representation of the data, and the last layer is a linear (or logistic) model applied to the last representation $\mathbf{h}^{(L-1)}$: the network *learns the representation* in which the final problem becomes simple, as we did by hand for the **XOR** problem.

Number of parameters. Since each unit of layer ℓ is connected to all the units of layer $\ell - 1$ plus a bias unit, the number of parameters of a fully connected network is

$$\sum_{\ell=1}^L d_{\ell} \times (d_{\ell-1} + 1).$$

In the example of Fig. 76, we have $d_0 = 3$, $d_1 = 3$, $d_2 = 2$ and $d_3 = 1$. There are $(3 + 1) \times 3 = 12$ parameters between the input layer and the first hidden layer, $(3 + 1) \times 2 = 8$ parameters between the two hidden layers, and $(2 + 1) \times 1 = 3$ parameters between the second hidden layer and the output layer, *i.e.* a total of 23 parameters. The network below has $(3 + 1) \times 3 \times 3 = 36$ parameters:



It has three outputs and can be used for *multi-class classification* with three classes; we explain how in Section 9.4. Modern networks have millions, or even billions, of parameters, often many more than the number of training examples.

9.2 Activation functions

Why do we need non-linear activations? Without activation functions (or with linear ones), a multilayer perceptron is useless, whatever its depth.

Proposition 9.2: Composition of affine maps

If all the activation functions $\sigma_1, \dots, \sigma_L$ are the identity, then the multilayer perceptron of Definition 9.1 is an affine function of its input: there exist $\mathbf{W} \in \mathbb{R}^{d_L \times d}$ and $\mathbf{b} \in \mathbb{R}^{d_L}$ such that $f_{\theta}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$ for all $\mathbf{x}$.

Proof. By induction on L : the result is clear for $L = 1$, and if $\mathbf{h}^{(L-1)} = \mathbf{W}'\mathbf{x} + \mathbf{b}'$, then $\mathbf{h}^{(L)} = \mathbf{W}^{(L)}(\mathbf{W}'\mathbf{x} + \mathbf{b}') + \mathbf{b}^{(L)} = (\mathbf{W}^{(L)}\mathbf{W}')\mathbf{x} + (\mathbf{W}^{(L)}\mathbf{b}' + \mathbf{b}^{(L)})$. $\square$

A deep linear network thus has exactly the same expressive power as a linear model: it cannot solve the **XOR** problem. The non-linearity of the activations is what makes the depth useful. Since the networks are trained with gradient-based methods, we also want activation functions that are differentiable (at least almost everywhere) with non-zero derivatives. The most common activation functions are given in Table 6 and represented in Fig. 77.

Name	$\sigma(z)$	$\sigma'(z)$	Range
Heaviside	$\mathbb{1}_{z \geq 0}$	0 (for $z \neq 0$)	$\{0, 1\}$
Sigmoid (logistic)	$\frac{1}{1 + e^{-z}}$	$\sigma(z)(1 - \sigma(z))$	$(0, 1)$
Hyperbolic tangent	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$1 - \tanh^2(z)$	$(-1, 1)$
ReLU	$\max(0, z)$	$\mathbb{1}_{z > 0}$ (for $z \neq 0$)	$[0, +\infty)$
Leaky ReLU	$\max(\beta z, z), 0 < \beta < 1$	1 if $z > 0$, β if $z < 0$	$\mathbb{R}$
ELU	z if $z > 0$, $e^z - 1$ otherwise	1 if $z > 0$, e^z otherwise	$(-1, +\infty)$

Table 6: Common activation functions and their derivatives.

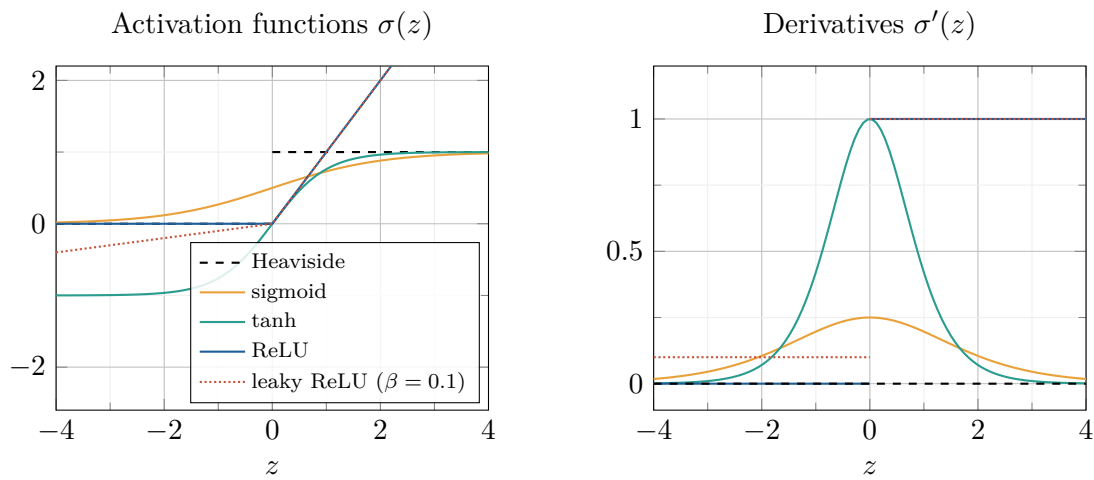


Figure 77: Graphs of the activation functions of [Table 6](#) (left) and of their derivatives (right). The derivative of the sigmoid is at most $1/4$ and those of the sigmoid and of tanh vanish when $|z|$ is large (saturation), whereas the derivative of the ReLU is equal to 1 on the whole positive half-line.

Sigmoid and hyperbolic tangent. The sigmoid $\sigma(z) = 1/(1+e^{-z})$ was the standard activation function until the 2000s. Its values lie in $(0, 1)$, so that it can be interpreted as a probability, which is why it is still used on the output layer for binary classification ([Section 9.4](#)). The hyperbolic tangent is a rescaled sigmoid, $\tanh(z) = 2\sigma(2z) - 1$, whose outputs are centered around 0, which usually makes the optimization easier. Both functions *saturate*: their derivatives tend to 0 when $|z| \rightarrow +\infty$, and the derivative of the sigmoid is bounded by $1/4$. We will see in [Section 9.5](#) that this causes the gradients to vanish in deep networks.

ReLU and its variants. The *Rectified Linear Unit* $\text{ReLU}(z) = \max(0, z)$ is now the default choice for hidden layers. It is very cheap to compute, it does not saturate for positive inputs (its derivative is equal to 1), and it produces sparse representations (many units are exactly equal to 0). It is not differentiable at 0, but this is not a problem in practice: we simply use 0 (or 1) as the derivative at this point. Its main drawback is that a unit whose pre-activation is negative for all the training examples has a zero gradient and no longer learns (a “dead” unit). The *leaky ReLU* and the *ELU* avoid this

issue by allowing a small non-zero slope for negative inputs. Smooth variants such as GELU or SiLU ($z \mapsto z \sigma(z)$) are popular in recent architectures.

The softmax function. The last activation function is not applied componentwise: it transforms a vector of scores into a probability distribution, and it is used on the output layer for multi-class classification.

Definition 9.2: Softmax function

The softmax function $\text{softmax} : \mathbb{R}^q \rightarrow \Delta_q = \{\mathbf{p} \in [0, 1]^q, \sum_k p_k = 1\}$ is defined by

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{l=1}^q e^{z_l}}, \quad k = 1, \dots, q.$$

Its Jacobian matrix is $\frac{\partial \text{softmax}(\mathbf{z})_k}{\partial z_l} = p_k (\mathbf{1}_{k=l} - p_l)$, where $\mathbf{p} = \text{softmax}(\mathbf{z})$, *i.e.* $J = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$.

The name comes from the fact that $\text{softmax}(\mathbf{z})$ is a smooth approximation of the indicator vector of $\arg \max_k z_k$. It is invariant to the addition of a constant to all the scores, $\text{softmax}(\mathbf{z} + c\mathbf{1}) = \text{softmax}(\mathbf{z})$, a property that is used in practice to avoid numerical overflows: we compute $\text{softmax}(\mathbf{z} - \max_k z_k \mathbf{1})$. For $q = 2$, the softmax reduces to the sigmoid: $\text{softmax}(\mathbf{z})_1 = \sigma(z_1 - z_2)$. We recover the multinomial logistic regression model of [Section 8](#).

9.3 Expressive power: the universal approximation theorem

We have seen that a network with one hidden layer can represent the **XOR** function. In fact, a network with one hidden layer can approximate *any* continuous function, provided that the hidden layer is large enough.

Theorem 9.3: Universal approximation (Cybenko, 1989; Hornik, 1991; Leshno et al., 1993)

Let $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be a continuous function which is not a polynomial (*e.g.* the sigmoid, tanh or ReLU), let $K \subset \mathbb{R}^d$ be a compact set and let $f^* : K \rightarrow \mathbb{R}$ be a continuous function. Then, for every $\varepsilon > 0$, there exist an integer N and parameters $\mathbf{w}_j \in \mathbb{R}^d$, $b_j, v_j \in \mathbb{R}$ ($j = 1, \dots, N$) such that the network with one hidden layer of N units

$$f(\mathbf{x}) = \sum_{j=1}^N v_j \sigma(\langle \mathbf{w}_j, \mathbf{x} \rangle + b_j) \quad \text{satisfies} \quad \sup_{\mathbf{x} \in K} |f(\mathbf{x}) - f^*(\mathbf{x})| \leq \varepsilon.$$

The proof relies on functional analysis (Hahn–Banach and Riesz representation theorems for Cybenko’s version) and is beyond the scope of these notes, but the intuition is easy to grasp in dimension one. With a sigmoid, the difference $\sigma(k(x-a)) - \sigma(k(x-b))$ of two hidden units is a “bump” that is close to the indicator function of $[a, b]$ when k

is large. A continuous function on $[0, 1]$ can be approximated by a piecewise constant function, *i.e.* by a weighted sum of such bumps, and thus by a network with one hidden layer, see Fig. 78.

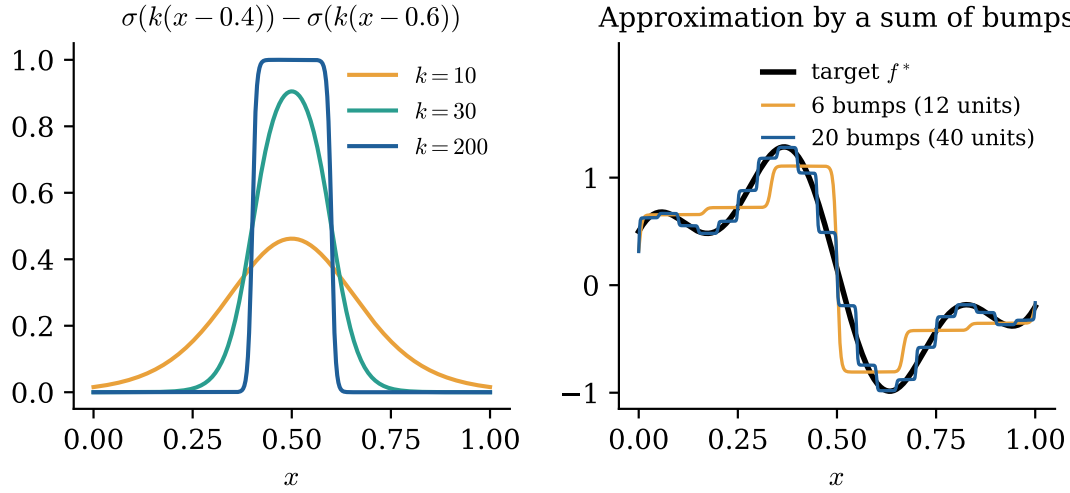


Figure 78: Intuition behind the universal approximation theorem. Left: the difference of two steep sigmoids is close to the indicator function of an interval. Right: a weighted sum of such bumps, *i.e.* a network with one hidden layer, approximates a continuous function; the approximation improves as the number of hidden units grows. [code: make_figures_nn.py:

fig_nn_universal]

This result must be read with care.

- It is an *existence* result: it says nothing about how to find the parameters, and gradient descent on a non-convex objective is not guaranteed to find them.
- The required number of hidden units N can be huge: in the construction above, it grows like $(1/\varepsilon)^d$, exponentially with the dimension.
- It says nothing about generalization: a model able to approximate any function is also able to fit the noise of the training set (see Section 6 on the estimation/approximation trade-off).

In practice, *deep* networks are preferred to wide and shallow ones: some functions can be represented by a deep network with a number of units that is exponentially smaller than the number required by a shallow network, and depth favors the learning of hierarchical representations (edges, then textures, then parts of objects, then objects for images). Figure 79 illustrates the effect of the size of the hidden layer on a two-dimensional problem.

9.4 Loss functions

Given a training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, the parameters of a network are learned by minimizing the (possibly regularized) empirical risk

$$\hat{L}_S(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i),$$

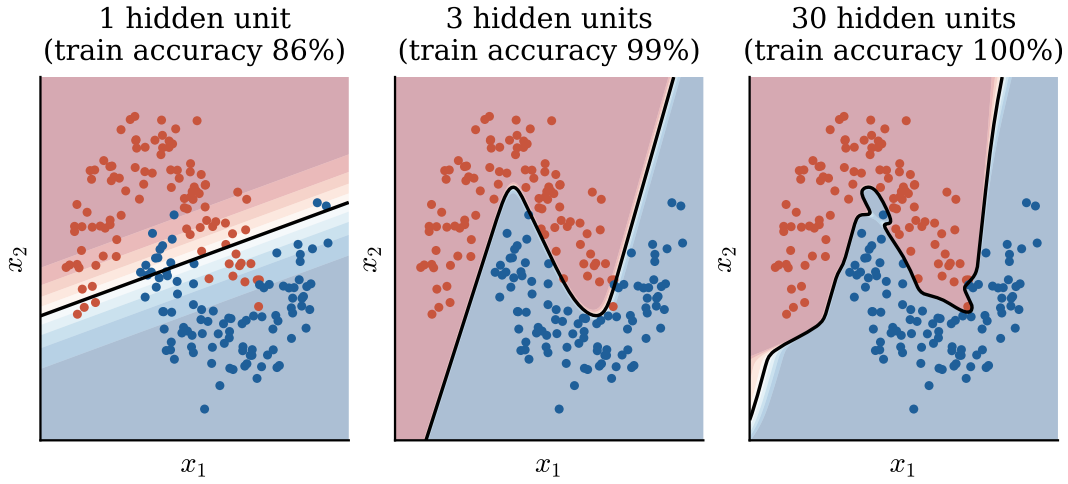


Figure 79: Decision boundaries of networks with one hidden layer (tanh activation, sigmoid output) trained on the same “two moons” dataset. With one hidden unit, the model is essentially a logistic regression; three units are enough to separate the two moons; with 30 units, the boundary starts to fit the noise of the training set. `[code: make_figures_nn.py: fig_nn_capacity]`

as in [Section 6](#). The choice of the loss ℓ goes together with the choice of the activation of the output layer. In each case below, we give the *derivative of the loss with respect to the pre-activation of the output layer* $\mathbf{a}^{(L)}$, which is the starting point of the backpropagation algorithm. We will see that it always takes the simple form “prediction minus target”.

Regression. For regression ($\mathcal{Y} = \mathbb{R}^q$), the output layer is linear ($\sigma_L = \text{id}$, so that $f_{\theta}(\mathbf{x}) = \mathbf{a}^{(L)}$) and we use the squared loss

$$\ell(f_{\theta}(\mathbf{x}), \mathbf{y}) = \frac{1}{2} \|\mathbf{f}_{\theta}(\mathbf{x}) - \mathbf{y}\|_2^2, \quad \text{hence} \quad \frac{\partial \ell}{\partial \mathbf{a}^{(L)}} = \mathbf{f}_{\theta}(\mathbf{x}) - \mathbf{y}.$$

As for linear regression, minimizing this loss amounts to maximizing the likelihood of the model $\mathbf{y} = \mathbf{f}_{\theta}(\mathbf{x}) + \varepsilon$ with a Gaussian noise $\varepsilon \sim \mathcal{N}(\mathbf{0}, s^2 \mathbf{I})$. The absolute loss or the Huber loss can be used when the data contain outliers.

Binary classification. For $\mathcal{Y} = \{0, 1\}$, the output layer has one unit with a sigmoid activation, and $p = f_{\theta}(\mathbf{x}) = \sigma(a^{(L)})$ is interpreted as an estimate of $\mathbb{P}(y = 1 \mid \mathbf{x})$. The negative log-likelihood of the Bernoulli model is the *binary cross-entropy*

$$\ell(p, y) = -y \ln p - (1 - y) \ln(1 - p), \quad \text{hence} \quad \frac{\partial \ell}{\partial a^{(L)}} = p - y,$$

where we used $\sigma' = \sigma(1 - \sigma)$. Written as a function of the score $a^{(L)}$ and with labels $\tilde{y} = 2y - 1 \in \{-1, +1\}$, this is exactly the logistic loss $\ln(1 + e^{-\tilde{y}a^{(L)}})$ of [Section 8](#). In particular, *a network without hidden layer, with a sigmoid output and the cross-entropy loss, is a logistic regression*: an MLP is a logistic regression applied to a learned representation $\mathbf{h}^{(L-1)}$ of the data.

Multi-class classification. For $\mathcal{Y} = \{1, \dots, q\}$, the output layer has q units and a softmax activation, and $p_k = \text{softmax}(\mathbf{a}^{(L)})_k$ estimates $\mathbb{P}(y = k \mid \mathbf{x})$. We encode the

label y by its *one-hot* vector $\mathbf{e}_y \in \{0,1\}^q$ (the y -th vector of the canonical basis) and use the *cross-entropy* loss, *i.e.* the negative log-likelihood of the categorical model:

$$\ell(\mathbf{p}, y) = -\ln p_y = -\sum_{k=1}^q (\mathbf{e}_y)_k \ln p_k.$$

This is the multinomial logistic regression model of [Section 8](#), applied to the last hidden representation.

Proposition 9.4: Gradient of the softmax cross-entropy

Let $\mathbf{z} \in \mathbb{R}^q$, $\mathbf{p} = \text{softmax}(\mathbf{z})$ and $y \in \{1, \dots, q\}$. Then

$$\ell = -\ln p_y = -z_y + \ln \sum_{l=1}^q e^{z_l} \quad \text{and} \quad \nabla_{\mathbf{z}} \ell = \mathbf{p} - \mathbf{e}_y.$$

Proof. By definition of the softmax,

$$\begin{aligned} \ell &= -\ln \frac{e^{z_y}}{\sum_l e^{z_l}} \\ &\downarrow \text{property of the logarithm} \\ &= -z_y + \ln \sum_{l=1}^q e^{z_l}. \end{aligned}$$

Differentiating with respect to z_k ,

$$\begin{aligned} \frac{\partial \ell}{\partial z_k} &= -\mathbb{1}_{k=y} + \frac{e^{z_k}}{\sum_l e^{z_l}} \\ &\downarrow \text{definition of the softmax} \\ &= p_k - (\mathbf{e}_y)_k. \end{aligned}$$

□

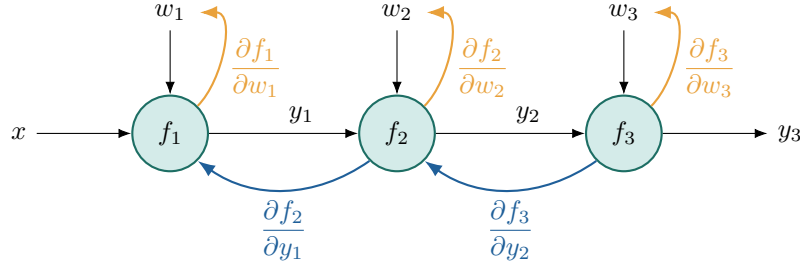
The function $\mathbf{z} \mapsto \ln \sum_l e^{z_l}$ (called *log-sum-exp*) is convex, so that the cross-entropy is a convex function of the scores. However, the scores $\mathbf{a}^{(L)}$ are themselves non-linear functions of the parameters, and the empirical risk of a network with at least one hidden layer is *not convex* in $\boldsymbol{\theta}$. The optimization algorithms of [Section 3](#) are thus only guaranteed to find stationary points, and the result of the training depends on the initialization. Surprisingly, in large networks, the local minima found by gradient methods are usually good enough.

Pitfall Never compute the cross-entropy by first computing $\mathbf{p} = \text{softmax}(\mathbf{z})$ and then $\ln p_y$: if p_y is rounded to 0, the loss is infinite. Use the expression $-z_y + \ln \sum_l e^{z_l}$ with the “max trick” of the previous section. This is what deep learning libraries do when the loss is fed with the scores (the *logits*) rather than with the probabilities.

9.5 Training: forward pass and backpropagation

A first example with the chain rule. To minimize $\hat{L}_S(\boldsymbol{\theta})$ with a gradient method, we need its gradient with respect to *all* the parameters of the network. Let us start

with a very simple network made of three layers with a single unit each, and a one-dimensional input¹⁵:



Here w_i and f_i respectively denote the parameter and the function computed by the i -th layer, and $y_i = f_i(y_{i-1}, w_i)$ its output, with $y_0 = x$. The training alternates two steps. The *forward* step feeds the network with data, one example at a time (stochastic) or a small subset of the data (a mini-batch), and computes the successive outputs y_1, y_2, y_3 and the value of the loss. The *backward* step computes the gradient of the loss with respect to each parameter and updates the parameters. The main difficulty is to propagate the error through the whole network: the parameter of the last layer is directly linked to the output, but this is not the case for the parameters of the first layers, whose influence on the output goes through all the following layers. The derivatives are obtained with the chain rule: *if g is differentiable at x and f is differentiable at $g(x)$, then*

$$(f \circ g)'(x) = f'(g(x)) \times g'(x).$$

For the sake of simplicity, we compute the derivatives of the output y_3 with respect to w_1, w_2 and w_3 ; the derivatives of the loss $\ell(y_3, y)$ are obtained by multiplying them by $\partial \ell / \partial y_3$.

- Derivative with respect to w_3 :

$$\frac{\partial y_3}{\partial w_3} = \frac{\partial f_3}{\partial w_3}(y_2, w_3).$$

- Derivative with respect to w_2 (which influences y_3 through y_2 only):

$$\begin{aligned} \frac{\partial y_3}{\partial w_2} &= \frac{\partial f_3}{\partial y_2}(y_2, w_3) \times \frac{\partial y_2}{\partial w_2} \\ &\downarrow \text{since } y_2 = f_2(y_1, w_2) \\ &= \frac{\partial f_3}{\partial y_2}(y_2, w_3) \times \frac{\partial f_2}{\partial w_2}(y_1, w_2). \end{aligned}$$

- Derivative with respect to w_1 (which influences y_3 through y_1 , then y_2):

$$\begin{aligned} \frac{\partial y_3}{\partial w_1} &= \frac{\partial f_3}{\partial y_2}(y_2, w_3) \times \frac{\partial y_2}{\partial w_1} \\ &\downarrow \text{chain rule again, } y_2 = f_2(y_1, w_2) \\ &= \frac{\partial f_3}{\partial y_2}(y_2, w_3) \times \frac{\partial f_2}{\partial y_1}(y_1, w_2) \times \frac{\partial y_1}{\partial w_1} \end{aligned}$$

¹⁵This example is taken from the PhD thesis of [Damien Fourure](#).

$$\begin{aligned} & \downarrow \text{ since } y_1 = f_1(x, w_1) \\ & = \underbrace{\frac{\partial f_3}{\partial y_2}(y_2, w_3) \times \frac{\partial f_2}{\partial y_1}(y_1, w_2)}_{\text{chain rule}} \times \frac{\partial f_1}{\partial w_1}(x, w_1). \end{aligned}$$

Two observations are essential. First, the derivatives involve the intermediate values y_1, y_2 : they must be computed and stored during the forward step. Second, the term underlined in blue, $\partial y_3 / \partial y_1$, also appears (partially) in the computation of $\partial y_3 / \partial w_2$: if we compute the derivatives *from the output to the input*, each product can reuse the previous one. Updating the weights by computing the derivatives backward, layer by layer, with the chain rule is called **backpropagation** (Rumelhart et al., 1986).

Backpropagation for a multilayer perceptron. Let us now derive the algorithm for the general multilayer perceptron of Definition 9.1, in matrix form. We consider a single example $(\mathbf{x}, \mathbf{y})$ and denote by $J = \ell(f_\theta(\mathbf{x}), \mathbf{y})$ its loss (we use the letter J since ℓ is used here as a layer index); the gradient of the empirical risk is the average of the gradients over the examples. The key quantity is the gradient of the loss with respect to the pre-activations of layer ℓ (the “error” of the layer):

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial J}{\partial \mathbf{a}^{(\ell)}} \in \mathbb{R}^{d_\ell}.$$

We denote by $\odot$ the componentwise (Hadamard) product of two vectors.

Proposition 9.5: Backpropagation equations

For a multilayer perceptron with componentwise activations on the hidden layers,

1. output layer: $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{a}^{(L)}} J$, e.g. $\boldsymbol{\delta}^{(L)} = f_\theta(\mathbf{x}) - \mathbf{y}$ for the three pairs (output activation, loss) of Section 9.4;
2. for $\ell = L - 1, \dots, 1$: $\boldsymbol{\delta}^{(\ell)} = \sigma'_\ell(\mathbf{a}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)\top} \boldsymbol{\delta}^{(\ell+1)})$;
3. for $\ell = 1, \dots, L$: $\frac{\partial J}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \mathbf{h}^{(\ell-1)\top} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ and $\frac{\partial J}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}$.

Proof. Point 1 was established in Section 9.4. For point 2, J depends on $a_j^{(\ell)}$ only through $h_j^{(\ell)} = \sigma_\ell(a_j^{(\ell)})$, which in turn influences all the pre-activations of the next layer, $a_k^{(\ell+1)} = \sum_{j'} W_{kj'}^{(\ell+1)} h_{j'}^{(\ell)} + b_k^{(\ell+1)}$. Hence

$$\begin{aligned} \delta_j^{(\ell)} &= \frac{\partial J}{\partial a_j^{(\ell)}} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial J}{\partial a_k^{(\ell+1)}} \frac{\partial a_k^{(\ell+1)}}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial a_j^{(\ell)}} \\ &\downarrow \text{ multivariate chain rule} \\ &= \sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1)} W_{kj}^{(\ell+1)} \sigma'_\ell(a_j^{(\ell)}) \\ &\downarrow \text{ definition of } \boldsymbol{\delta}^{(\ell+1)} \text{ and of } \mathbf{a}^{(\ell+1)} \\ &= \sigma'_\ell(a_j^{(\ell)}) \left(\mathbf{W}^{(\ell+1)\top} \boldsymbol{\delta}^{(\ell+1)} \right)_j. \end{aligned}$$

Algorithm 5: Training of a multilayer perceptron with backpropagation and mini-batch gradient descent

Input: training set $S = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$, architecture $(d_\ell, \sigma_\ell)_{\ell=1}^L$, learning rate η , batch size B , number of epochs T

Output: parameters $\theta = (\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)})_{\ell=1}^L$

- 1 Initialize the weights randomly (Section 9.6) and the biases to $\mathbf{0}$;
- 2 **for** $t = 1, \dots, T$ **do**
- 3 Shuffle S and split it into mini-batches of size B ;
- 4 **for** each mini-batch $\mathcal{B}$ **do**
- 5 **for** each $(\mathbf{x}, \mathbf{y}) \in \mathcal{B}$ **do**
- 6 // forward pass: compute and store the activations
- 7 $\mathbf{h}^{(0)} \leftarrow \mathbf{x}$;
- 8 **for** $\ell = 1, \dots, L$ **do**
- 9 $\mathbf{a}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} \mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$ and $\mathbf{h}^{(\ell)} \leftarrow \sigma_\ell(\mathbf{a}^{(\ell)})$;
- 10 // backward pass: propagate the errors
- 11 $\delta^{(L)} \leftarrow \nabla_{\mathbf{a}^{(L)}} \ell(\mathbf{h}^{(L)}, \mathbf{y})$;
- 12 **for** $\ell = L, \dots, 1$ **do**
- 13 $\mathbf{g}_{\mathbf{W}^{(\ell)}}(\mathbf{x}) \leftarrow \delta^{(\ell)} \mathbf{h}^{(\ell-1)\top}$ and $\mathbf{g}_{\mathbf{b}^{(\ell)}}(\mathbf{x}) \leftarrow \delta^{(\ell)}$;
- 14 **if** $\ell > 1$ **then** $\delta^{(\ell-1)} \leftarrow \sigma'_{\ell-1}(\mathbf{a}^{(\ell-1)}) \odot (\mathbf{W}^{(\ell)\top} \delta^{(\ell)})$;
- // update of the parameters
- for** $\ell = 1, \dots, L$ **do**
- $\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \frac{\eta}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{g}_{\mathbf{W}^{(\ell)}}(\mathbf{x})$ and $\mathbf{b}^{(\ell)} \leftarrow \mathbf{b}^{(\ell)} - \frac{\eta}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{g}_{\mathbf{b}^{(\ell)}}(\mathbf{x})$;

For point 3, the weight $W_{jk}^{(\ell)}$ only influences $a_j^{(\ell)} = \sum_{k'} W_{jk'}^{(\ell)} h_{k'}^{(\ell-1)} + b_j^{(\ell)}$, so that

$$\frac{\partial J}{\partial W_{jk}^{(\ell)}} = \frac{\partial J}{\partial a_j^{(\ell)}} \frac{\partial a_j^{(\ell)}}{\partial W_{jk}^{(\ell)}} = \delta_j^{(\ell)} h_k^{(\ell-1)} \quad \text{and} \quad \frac{\partial J}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)}. \quad \square$$

The error is thus propagated *backward* through the transposed weight matrices, which gives its name to the algorithm. The gradient with respect to a weight is the product of the error at its “output end” and of the activation at its “input end”. The complete training procedure with mini-batch gradient descent is given in Algorithm 5. For a mini-batch B of examples, the vectors $\mathbf{h}^{(\ell)}$ and $\delta^{(\ell)}$ become matrices of size $d_\ell \times |B|$ (one column per example), and all the operations are matrix products, which run very efficiently on GPUs.

Example 9.1 (One step of backpropagation by hand). Consider a network with $d = 2$ inputs, one hidden layer of 2 ReLU units and a linear output unit, trained with the squared loss $\ell = \frac{1}{2}(f - y)^2$ on the single example $\mathbf{x} = (1, 2)^\top$, $y = 2$. The parameters are

$$\mathbf{W}^{(1)} = \begin{pmatrix} 1 & -1 \\ 0.5 & 0.5 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0.5 \\ -1 \end{pmatrix}, \quad \mathbf{W}^{(2)} = \begin{pmatrix} 1 & 2 \end{pmatrix}, \quad b^{(2)} = 0.$$

Forward pass. $\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} = (1 - 2 + 0.5, 0.5 + 1 - 1)^\top = (-0.5, 0.5)^\top$, so $\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{a}^{(1)}) = (0, 0.5)^\top$. Then $f = a^{(2)} = 1 \times 0 + 2 \times 0.5 + 0 = 1$ and

$$\ell = \frac{1}{2}(1 - 2)^2 = 0.5.$$

Backward pass. For the output layer, $\delta^{(2)} = f - y = -1$, hence

$$\frac{\partial \ell}{\partial \mathbf{W}^{(2)}} = \delta^{(2)} \mathbf{h}^{(1)\top} = (0, -0.5) \quad \text{and} \quad \frac{\partial \ell}{\partial b^{(2)}} = -1.$$

For the hidden layer, $\mathbf{W}^{(2)\top} \delta^{(2)} = (-1, -2)^\top$ and $\text{ReLU}'(\mathbf{a}^{(1)}) = (0, 1)^\top$, hence $\boldsymbol{\delta}^{(1)} = (0, -2)^\top$ and

$$\frac{\partial \ell}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}^{(1)} \mathbf{x}^\top = \begin{pmatrix} 0 & 0 \\ -2 & -4 \end{pmatrix} \quad \text{and} \quad \frac{\partial \ell}{\partial \mathbf{b}^{(1)}} = \begin{pmatrix} 0 \\ -2 \end{pmatrix}.$$

Update with $\eta = 0.05$: $\mathbf{W}^{(1)} \leftarrow \begin{pmatrix} 1 & -1 \\ 0.6 & 0.7 \end{pmatrix}$, $\mathbf{b}^{(1)} \leftarrow (0.5, -0.9)^\top$, $\mathbf{W}^{(2)} \leftarrow (1, 2.025)$ and $b^{(2)} \leftarrow 0.05$.

New forward pass. $\mathbf{a}^{(1)} = (-0.5, 1.1)^\top$, $\mathbf{h}^{(1)} = (0, 1.1)^\top$ and $f = 2.025 \times 1.1 + 0.05 = 2.2775$: the loss decreased from 0.5 to $\frac{1}{2}(0.2775)^2 \approx 0.039$. Note that the first hidden unit is inactive ($a_1^{(1)} < 0$): its incoming weights receive a zero gradient and are not modified. If this happens for all the examples, the unit is “dead”.

Computational cost. The forward pass of a fully connected network costs about $\sum_\ell d_\ell d_{\ell-1}$ multiplications per example, *i.e.* of the order of the number of parameters P . The backward pass requires two matrix products per layer (one for $\boldsymbol{\delta}^{(\ell-1)}$, one for the gradient of $\mathbf{W}^{(\ell)}$), so its cost is about twice the cost of the forward pass. Computing the full gradient thus costs $O(P)$ operations per example, *the same order as evaluating the network*. Compare with a finite-difference approximation $\frac{\partial \ell}{\partial \theta_j} \approx \frac{\ell(\boldsymbol{\theta} + \epsilon \mathbf{e}_j) - \ell(\boldsymbol{\theta})}{\epsilon}$, which requires one forward pass per parameter, *i.e.* $O(P^2)$ operations: backpropagation is what makes the training of large networks possible. The price to pay is memory, since all the activations of the forward pass must be stored until the backward pass. Backpropagation is a particular case of *reverse-mode automatic differentiation*: modern libraries (PyTorch, TensorFlow, JAX) build the computational graph of any function made of elementary operations and compute its gradient automatically, so that we never have to derive the backpropagation equations by hand in practice. Finite differences remain useful to *check* a hand-written gradient (*gradient checking*).

Vanishing and exploding gradients. Unrolling the recursion of Proposition 9.5 shows that the error at layer ℓ is a product of $L - \ell$ terms:

$$\boldsymbol{\delta}^{(\ell)} = D^{(\ell)} \mathbf{W}^{(\ell+1)\top} D^{(\ell+1)} \mathbf{W}^{(\ell+2)\top} \dots D^{(L-1)} \mathbf{W}^{(L)\top} \boldsymbol{\delta}^{(L)},$$

with $D^{(k)} = \text{diag}(\sigma'_k(\mathbf{a}^{(k)}))$, so that

$$\|\boldsymbol{\delta}^{(\ell)}\| \leq \prod_{k=\ell+1}^L \left(\|\mathbf{W}^{(k)}\| \max_z |\sigma'_{k-1}(z)| \right) \|\boldsymbol{\delta}^{(L)}\|.$$

If the factors are smaller than 1, the gradient of the first layers decreases exponentially fast with the depth: this is the *vanishing gradient* problem, and the first layers barely learn. With the sigmoid, $|\sigma'| \leq 1/4$, and this is precisely what happens unless

the weights are large. Conversely, if the factors are larger than 1, the gradients can grow exponentially (*exploding gradients*) and the training diverges. This phenomenon, identified by [Bengio et al. \(1994\)](#) for recurrent networks, long prevented the training of deep networks. The remedies, presented in the next subsections, are: non-saturating activations (ReLU), a careful initialization, normalization layers, residual connections and, for exploding gradients, *gradient clipping* (rescaling the gradient when its norm exceeds a threshold).

9.6 Optimization

Stochastic and mini-batch gradient descent. The empirical risk is an average over m examples, and m can be of the order of millions. Computing the full gradient $\nabla \hat{L}_S(\boldsymbol{\theta})$ at each iteration, as in the gradient descent of [Section 3](#), is too costly. *Stochastic gradient descent* (SGD) replaces it by the gradient on one example i_t drawn uniformly at random:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{i_t}), y_{i_t}).$$

This direction is an *unbiased* estimate of the full gradient, $\mathbb{E}_{i_t} [\nabla_{\boldsymbol{\theta}} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{i_t}), y_{i_t})] = \nabla \hat{L}_S(\boldsymbol{\theta}_t)$, but it is noisy. In practice, we use *mini-batch* gradient descent: the gradient is averaged over a small batch $\mathcal{B}_t$ of B examples (typically $B \in \{32, \dots, 512\}$), which divides the variance of the estimate by B and exploits the parallelism of GPUs ([Algorithm 5](#)). One pass over the whole training set is called an *epoch*. For convex problems, SGD converges if the step sizes satisfy $\sum_t \eta_t = +\infty$ and $\sum_t \eta_t^2 < +\infty$ (e.g. $\eta_t \propto 1/t$). For neural networks, the noise of SGD is also believed to help escape saddle points and poor local minima.

Momentum. When the objective is ill-conditioned (a narrow valley), gradient descent oscillates across the valley and progresses slowly along it. The *momentum* method ([Polyak, 1964](#)) (or “heavy ball”) accumulates the past gradients in a velocity $\mathbf{v}$:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \mathbf{g}_t, \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{v}_{t+1},$$

where $\mathbf{g}_t$ is the (stochastic) gradient at $\boldsymbol{\theta}_t$ and $\beta \in [0, 1)$ (typically 0.9). The oscillating components of the gradient cancel out in the average, while the consistent ones add up. The *Nesterov* variant evaluates the gradient at the “look-ahead” point $\boldsymbol{\theta}_t - \eta\beta\mathbf{v}_t$.

Adaptive methods: RMSProp and Adam. Different parameters may require different step sizes. *RMSProp* divides the step of each coordinate by a running average of the squared gradients:

$$\mathbf{s}_{t+1} = \rho \mathbf{s}_t + (1 - \rho) \mathbf{g}_t \odot \mathbf{g}_t, \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbf{s}_{t+1}} + \epsilon} \odot \mathbf{g}_t,$$

where the square root and the division are componentwise, $\rho \approx 0.9$ and $\epsilon \approx 10^{-8}$ avoids divisions by zero. *Adam* ([Kingma and Ba, 2015](#)) combines momentum and RMSProp, with a bias correction for the first iterations (the averages are initialized at 0):

$$\begin{aligned} \mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t, & \hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1 - \beta_1^t}, \\ \mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t, & \hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1 - \beta_2^t}, \end{aligned}$$

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon},$$

with the default values $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 10^{-3}$. Adam is the default optimizer of many practitioners because it is robust to the choice of η , although well-tuned SGD with momentum sometimes generalizes better. The behavior of these methods is illustrated in Fig. 80.

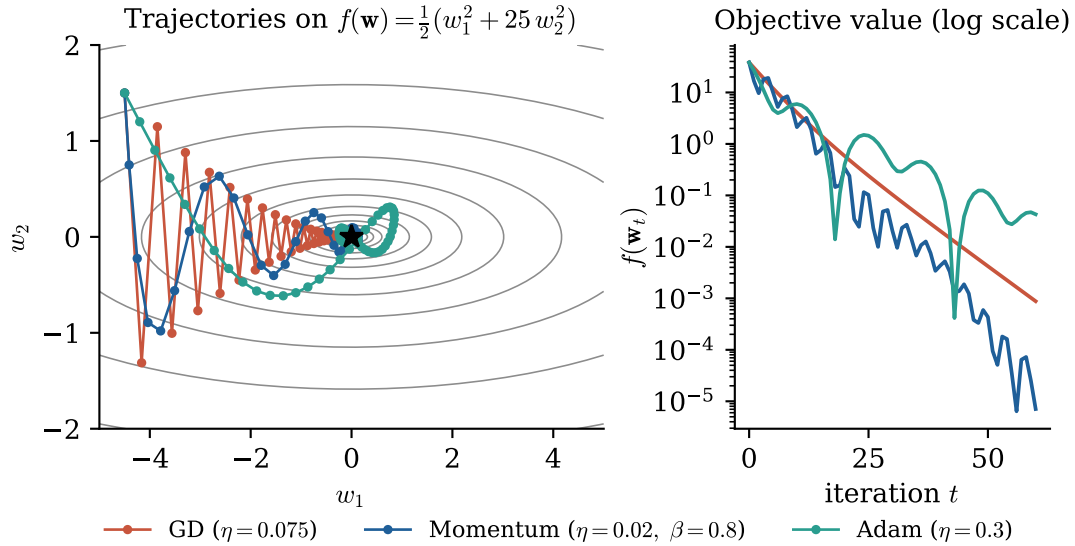


Figure 80: Gradient descent, momentum and Adam on an ill-conditioned quadratic function (deterministic gradients, 60 iterations from the same starting point). Gradient descent oscillates along the steep direction and progresses slowly along the flat one; momentum damps the oscillations; Adam normalizes the step of each coordinate and quickly reaches a neighborhood of the minimum, but with a constant learning rate it keeps oscillating around it. [code:

make_figures_nn.py: fig_nn_optimizers]

Learning-rate schedules. The learning rate η is the most important hyperparameter. Too large, the training diverges; too small, it is very slow. In practice, the learning rate is decreased during the training: step decay (divide η by 10 every few epochs), exponential decay $\eta_t = \eta_0 \gamma^t$, inverse decay $\eta_t = \eta_0 / (1 + \kappa t)$, or cosine annealing $\eta_t = \frac{\eta_0}{2} (1 + \cos(\pi t / T))$. Large models are often trained with a short *warm-up* phase, during which η increases linearly from 0 to η_0 .

Weight initialization. The initialization matters for a non-convex problem. The weights cannot be initialized to $\mathbf{0}$ (or to any common value): all the units of a layer would then compute the same function and receive the same gradient, and they would remain identical forever. The weights are thus initialized randomly to *break the symmetry*, and their scale is chosen so that the variance of the activations (and of the gradients) is preserved from one layer to the next. Consider a unit $a_j = \sum_{k=1}^{n_{\text{in}}} W_{jk} h_k$ where the W_{jk} are independent, centered, with variance v , and independent of the inputs h_k , which have variance $\text{Var}(h)$ and zero mean. Then

$$\text{Var}(a_j) = \sum_{k=1}^{n_{\text{in}}} \text{Var}(W_{jk} h_k) = n_{\text{in}} v \text{Var}(h).$$

Keeping $\text{Var}(a_j) = \text{Var}(h)$ requires $v = 1/n_{\text{in}}$ (for an activation close to the identity around 0, such as $\tanh$), while the same reasoning on the backward pass requires $v = 1/n_{\text{out}}$. This leads to the following classical choices.

- *Xavier/Glorot initialization* (Glorot and Bengio, 2010), for $\tanh$ or sigmoid: $v = \frac{2}{n_{\text{in}} + n_{\text{out}}}$, e.g. $W_{jk} \sim \mathcal{U}\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]$.
- *He initialization* (He et al., 2015), for ReLU: $v = \frac{2}{n_{\text{in}}}$, e.g. $W_{jk} \sim \mathcal{N}(0, 2/n_{\text{in}})$; the factor 2 compensates for the fact that the ReLU sets half of its inputs to 0.

The biases are usually initialized to 0.

9.7 Regularization

Neural networks have many parameters and can easily overfit the training set (see Fig. 79). Besides collecting more data, several regularization techniques are available.

Weight decay. As for ridge regression, we add the penalty $\frac{\lambda}{2} \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_{\mathcal{F}}^2$ to the empirical risk (the biases are usually not penalized). The gradient step becomes

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \left(\nabla_{\mathbf{W}^{(\ell)}} \hat{L}_S + \lambda \mathbf{W}^{(\ell)} \right) = (1 - \eta\lambda) \mathbf{W}^{(\ell)} - \eta \nabla_{\mathbf{W}^{(\ell)}} \hat{L}_S,$$

i.e. the weights are shrunk by a factor $(1 - \eta\lambda)$ at each step, hence the name *weight decay*.

Early stopping. During the training, the training loss decreases while the validation loss first decreases, then increases when the network starts to overfit (Fig. 81). *Early stopping* monitors the validation loss after each epoch and returns the parameters with the lowest validation loss; the training is stopped when this loss has not improved for a given number of epochs (the *patience*). It is simple, cheap and very effective: the number of epochs acts as a regularization hyperparameter, and for a quadratic loss, early stopping can be shown to be closely related to ℓ_2 regularization.

Dropout. *Dropout* (Srivastava et al., 2014) randomly “switches off” units during the training. At each forward pass and for each hidden unit independently, the activation is multiplied by a Bernoulli variable $r_j \sim \mathcal{B}(1 - p)$, where p is the dropout rate (typically $p \in [0.1, 0.5]$). In the “inverted dropout” implementation, the kept activations are divided by $1 - p$, so that $\mathbb{E}[r_j h_j / (1 - p)] = h_j$, and nothing has to be changed at test time, when all the units are kept. Dropout prevents the units from co-adapting, since a unit cannot rely on the presence of the others, and it can be interpreted as training an ensemble of exponentially many sub-networks sharing their weights (see the ensemble methods of Section 10). For a linear model trained with the squared loss, dropout on the inputs is, in expectation, equivalent to an ℓ_2 penalty; more generally, it acts as an adaptive, data-dependent regularization (Wager et al., 2013).

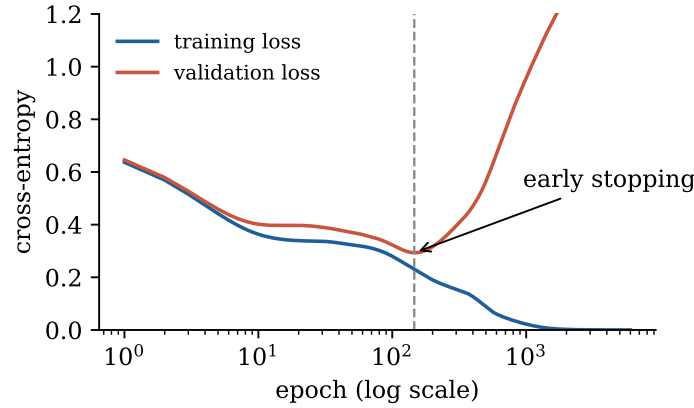


Figure 81: Training and validation losses of an over-parameterized network (60 hidden units, 60 noisy training examples). The validation loss reaches its minimum after about 150 epochs, then increases while the training loss keeps decreasing: the network overfits. [code: `make_figures_nn.py: fig_nn_early_stopping`]

Data augmentation. The best regularization is more data. When the task is invariant to some transformations, we can create new training examples by applying them to the existing ones: for images, random crops, horizontal flips, small rotations, changes of color or contrast; for sounds, time shifts or added noise. This encodes prior knowledge about the problem, and it is essential to train large vision models.

Batch normalization. *Batch normalization* (Ioffe and Szegedy, 2015) normalizes the pre-activations of a layer over the current mini-batch $\mathcal{B}$. For each unit j ,

$$\hat{a}_j = \frac{a_j - \mu_{\mathcal{B},j}}{\sqrt{s_{\mathcal{B},j}^2 + \epsilon}}, \quad \tilde{a}_j = \gamma_j \hat{a}_j + \beta_j,$$

where $\mu_{\mathcal{B},j}$ and $s_{\mathcal{B},j}^2$ are the empirical mean and variance of a_j over the mini-batch, and γ_j, β_j are learned parameters (which allow the network to recover the original activation if needed). At test time, the batch statistics are replaced by running averages computed during the training. Batch normalization stabilizes and accelerates the training, allows larger learning rates, makes the network less sensitive to the initialization and has a slight regularizing effect. Its variant *layer normalization*, which normalizes over the units of a layer rather than over the examples, is used in recurrent networks and transformers.

9.8 Convolutional neural networks

Motivation. Fully connected networks are not suited to images. A color image of size 224×224 has $224 \times 224 \times 3 \approx 150\,000$ entries: a single fully connected layer of 1 000 units would already have 150 million parameters. Moreover, such a layer ignores the structure of the image: the pixels are treated as an unordered set of features, and an object must be learned again at each position in the image. *Convolutional neural networks* (CNNs) (LeCun et al., 1998) exploit two properties of images:

- *locality*: the relevant patterns (edges, corners, textures) are local, so that each unit only needs to look at a small neighborhood of pixels (its *receptive field*);

- *translation invariance*: a pattern has the same meaning wherever it appears, so that the same weights can be used at every position (*weight sharing*).

The convolution operation. Let $\mathbf{I} \in \mathbb{R}^{n \times n}$ be an input image (one channel) and $\mathbf{K} \in \mathbb{R}^{k \times k}$ a *filter* (or kernel), with k small (typically 3 or 5). The (discrete) convolution¹⁶ produces the *feature map*

$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_{u=1}^k \sum_{v=1}^k I_{i+u-1, j+v-1} K_{u,v} + b,$$

i.e. the filter slides over the image, and at each position we compute the inner product between the filter and the patch of the image it covers, see Fig. 82. A convolution layer is thus a linear layer with a very particular weight matrix: sparse (locality) and with many shared entries (weight sharing). It is followed by a non-linear activation, usually a ReLU. A filter acts as a pattern detector: the feature map takes large values where the image looks like the filter. In classical image processing, filters are designed by hand (*e.g.* the Sobel filter of Fig. 83 detects vertical edges); in a CNN, they are learned by backpropagation.

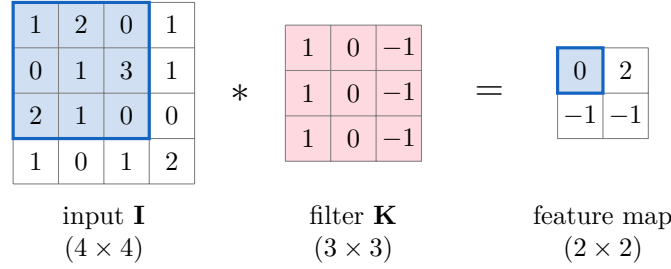


Figure 82: A convolution without padding and with stride 1. The highlighted output is the inner product between the filter and the highlighted 3×3 patch: $(1 + 0 + 2) \times 1 + (2 + 1 + 1) \times 0 + (0 + 3 + 0) \times (-1) = 0$. This filter responds to vertical transitions from bright (left) to dark (right).

Channels, padding and stride. A color image has $C = 3$ channels (red, green, blue), and the output of a convolution layer has as many channels as filters. A filter of a layer with C_{in} input channels is thus a tensor of size $k \times k \times C_{\text{in}}$, and a layer with C_{out} filters has $(k^2 C_{\text{in}} + 1) C_{\text{out}}$ parameters, *independently of the size of the image*. Two other hyperparameters control the size of the output:

- the *padding* p : we add p rows and columns of zeros around the input, *e.g.* to preserve the spatial size (with $p = (k - 1)/2$ for an odd k) and to treat the border pixels like the others;
- the *stride* s : the filter moves by s pixels at a time, which reduces the size of the output.

For an input of size $n \times n$, the output of a convolution with a $k \times k$ filter, padding p and stride s is of size $n' \times n'$ with

$$n' = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1.$$

¹⁶Strictly speaking, this operation is a cross-correlation: the mathematical convolution flips the kernel. Since the kernel is learned, the distinction does not matter.

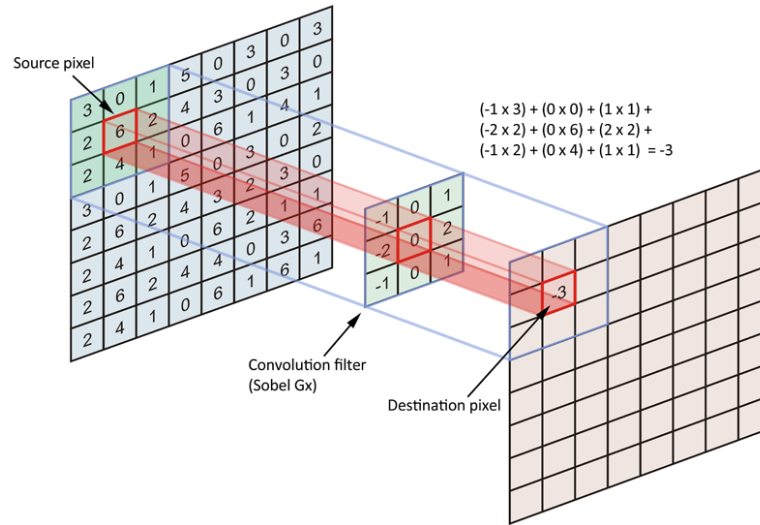


Figure 83: A convolution with the 3×3 Sobel filter G_x , a classical hand-designed filter that detects vertical edges. Each pixel of the output (destination pixel) is the inner product of the filter with the neighborhood of the corresponding input pixel (source pixel).

For instance, in Fig. 82, $n' = (4 + 0 - 3)/1 + 1 = 2$.

Pooling. A *pooling* layer reduces the spatial size of the feature maps by summarizing each window (typically 2×2 with stride 2) by its maximum (*max pooling*) or its average (*average pooling*), independently on each channel, see Fig. 84. It has no parameters. Pooling reduces the computational cost of the next layers, increases the receptive field of the units, and makes the representation robust to small translations of the input.

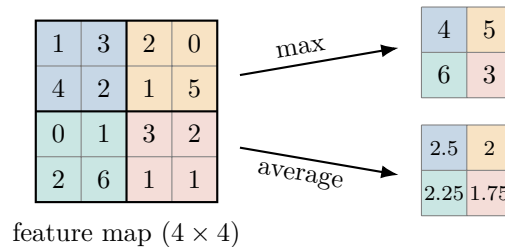


Figure 84: Max pooling (top) and average pooling (bottom) with a 2×2 window and stride 2: each colored block of the feature map is summarized by a single value.

A typical architecture. A classical CNN for image classification alternates blocks of the form $[\text{convolution} \rightarrow \text{ReLU}] \times n \rightarrow \text{pooling}$. Along the network, the spatial size of the feature maps decreases while the number of channels increases: the first layers detect simple patterns (edges, colors), and the deeper layers combine them into increasingly abstract patterns (textures, parts of objects, objects). The last feature maps are then flattened into a vector and fed to a few fully connected layers ending with a softmax. The first successful CNN, LeNet-5 (LeCun et al., 1998), recognized handwritten digits with two convolution layers; the victory of AlexNet at the ImageNet competition in 2012 marked the beginning of the deep learning era. Since then, the architectures have become deeper and deeper (VGG, Inception, ...), which was made

possible by the *residual connections* of ResNet (He et al., 2016): a block computes $\mathbf{h} \mapsto \mathbf{h} + F(\mathbf{h})$ instead of $F(\mathbf{h})$, so that the gradient can flow directly through the identity path, which greatly alleviates the vanishing gradient problem and allows the training of networks with more than 100 layers. Figure 85 summarizes this evolution (Khan et al., 2020). CNNs are also used for other tasks than classification, such as object detection or semantic segmentation, where a class must be predicted for each pixel of the image (Fig. 86).

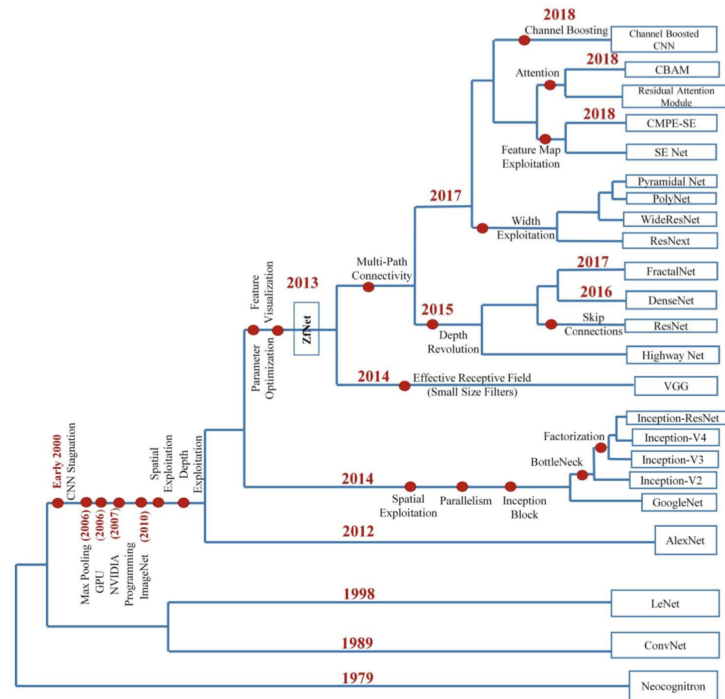


Figure 85: A taxonomy of the main CNN architectures, from the Neocognitron (1979) and LeNet (1998) to the architectures of 2018; figure from Khan et al. (2020).



Figure 86: Semantic segmentation of a street scene: a convolutional network predicts the class of each pixel (car, pedestrian, road, sidewalk, vegetation, sign, ...). Result obtained with the GridNet architecture (Fourure et al., 2017).

9.9 Recurrent neural networks

Sequential data. Many data are sequences of variable length: texts (sequences of words), speech, time series, DNA sequences, user logs. A *recurrent neural network* (RNN) processes a sequence $\mathbf{x}_1, \dots, \mathbf{x}_T$ one element at a time, and maintains a *hidden state* $\mathbf{h}_t$ that summarizes the past:

$$\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}), \quad \hat{\mathbf{y}}_t = \mathbf{V}\mathbf{h}_t + \mathbf{c},$$

with $\mathbf{h}_0 = \mathbf{0}$. The same parameters $(\mathbf{W}, \mathbf{U}, \mathbf{b}, \mathbf{V}, \mathbf{c})$ are used at every time step (weight sharing through time), so that the model can process sequences of any length. Depending on the task, we use all the outputs $\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_T$ (*e.g.* tagging each word of a sentence) or only the last one (*e.g.* classifying the sentiment of a review). Figure 87 shows that an RNN *unrolled* through time is a deep feedforward network with shared weights. Its gradient is thus computed by backpropagation on the unrolled network, an algorithm called *backpropagation through time* (BPTT).

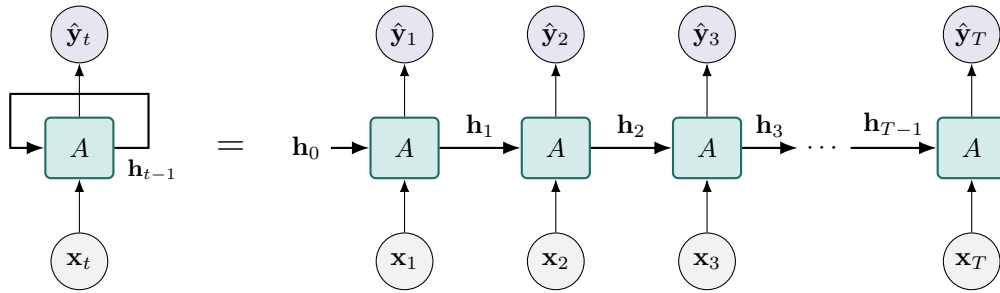


Figure 87: A recurrent neural network (left) and its unrolled version (right): the block A receives the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$, computes the new hidden state $\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$ and the output $\hat{\mathbf{y}}_t = \mathbf{V}\mathbf{h}_t + \mathbf{c}$. The same parameters are used in every copy of A .

Long-term dependencies. The gradient of the loss at time T with respect to the hidden state at time t involves the product of $T - t$ Jacobian matrices $\text{diag}(\tanh'(\cdot))\mathbf{W}$, with the *same* matrix $\mathbf{W}$. As explained in Section 9.5, this product vanishes or explodes exponentially fast with $T - t$, depending on the spectral norm of $\mathbf{W}$ (Bengio et al., 1994). In practice, a simple RNN is unable to learn dependencies that span more than a few dozen time steps. Gated architectures solve this issue.

LSTM. The *Long Short-Term Memory* network (Hochreiter and Schmidhuber, 1997) adds a *cell state* $\mathbf{c}_t$, a memory updated in an additive way and controlled by three *gates* with values in $(0, 1)$, see Fig. 88:

$$\begin{aligned} \mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f) && \text{(forget gate)} \\ \mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i) && \text{(input gate)} \\ \mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o) && \text{(output gate)} \\ \tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c) && \text{(candidate memory)} \\ \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \quad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \end{aligned}$$

The forget gate decides what to erase from the memory, the input gate what to write in it, and the output gate what to read from it. When $\mathbf{f}_t \approx \mathbf{1}$ and $\mathbf{i}_t \approx \mathbf{0}$, the memory

is copied unchanged and the gradient flows through time without vanishing, as in a residual connection.

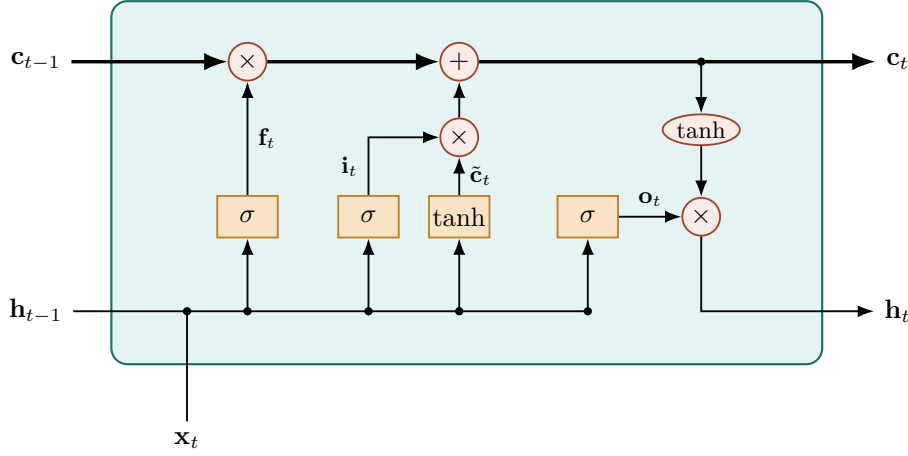


Figure 88: An LSTM cell. The thick upper line carries the cell state $\mathbf{c}_t$, which is only modified by a multiplication (forget gate $\mathbf{f}_t$) and an addition (input gate $\mathbf{i}_t$ times the candidate $\tilde{\mathbf{c}}_t$); the output gate $\mathbf{o}_t$ controls what is read from the memory to produce $\mathbf{h}_t$. The yellow boxes are layers with their own parameters, the red circles are componentwise operations.

GRU. The *Gated Recurrent Unit* (Choi et al., 2014) is a simplified version of the LSTM, with two gates and no separate cell state, see Fig. 89:

$$\begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{h}_{t-1} + \mathbf{U}_z \mathbf{x}_t + \mathbf{b}_z) && \text{(update gate)} \\ \mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{h}_{t-1} + \mathbf{U}_r \mathbf{x}_t + \mathbf{b}_r) && \text{(reset gate)} \\ \hat{\mathbf{h}}_t &= \tanh(\mathbf{W}_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{U}_h \mathbf{x}_t + \mathbf{b}_h) && \text{(candidate state)} \\ \mathbf{h}_t &= (\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \hat{\mathbf{h}}_t. \end{aligned}$$

It has fewer parameters than an LSTM and often performs similarly.

9.10 Other architectures

The building blocks presented so far can be assembled in many ways. We briefly present a few important architectures; see Goodfellow et al. (2016) for more details.

Autoencoders. An *autoencoder* is trained to reproduce its input: an *encoder* g_ϕ maps $\mathbf{x}$ to a code $\mathbf{z} = g_\phi(\mathbf{x})$ of small dimension, and a *decoder* f_θ reconstructs $\hat{\mathbf{x}} = f_\theta(\mathbf{z})$ from it, see Fig. 90. The two networks are trained jointly by minimizing the reconstruction error $\sum_i \|\mathbf{x}_i - f_\theta(g_\phi(\mathbf{x}_i))\|^2$, which does not require any label: this is unsupervised learning. The bottleneck forces the network to keep only the most important information: an autoencoder performs a non-linear dimensionality reduction (Hinton and Salakhutdinov, 2006), and a linear autoencoder with the squared loss recovers the subspace of principal component analysis. Autoencoders are also used for denoising (the network reconstructs $\mathbf{x}$ from a corrupted version) and for anomaly detection (anomalies are poorly reconstructed).

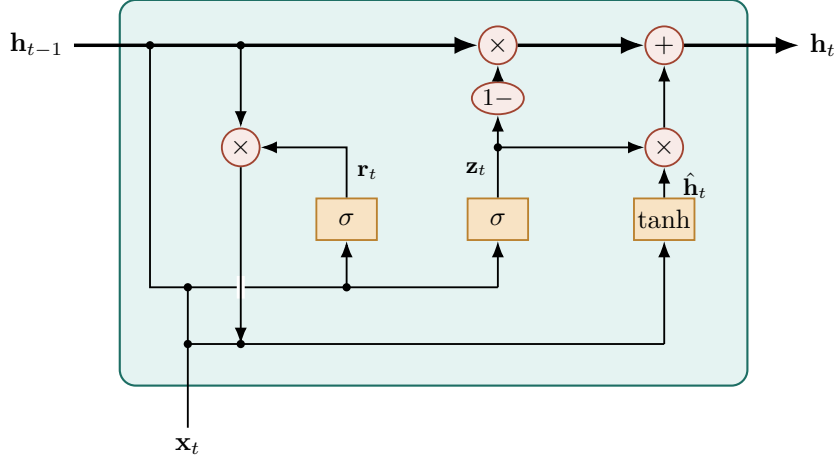


Figure 89: A GRU cell: the update gate $\mathbf{z}_t$ interpolates between the previous state $\mathbf{h}_{t-1}$ (upper line) and the candidate state $\hat{\mathbf{h}}_t$; the reset gate $\mathbf{r}_t$ controls how much of the past is used to compute the candidate. The yellow boxes are layers with their own parameters, the red circles are componentwise operations.

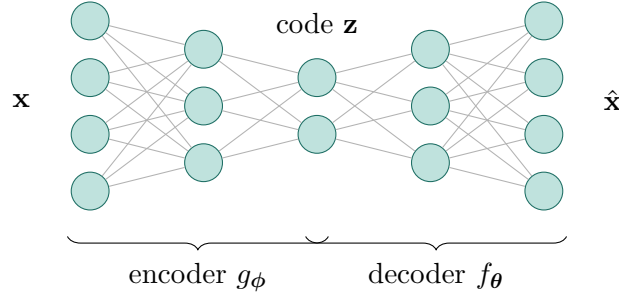


Figure 90: An autoencoder: the encoder compresses the input $\mathbf{x}$ into a low-dimensional code $\mathbf{z}$, and the decoder reconstructs $\hat{\mathbf{x}} \approx \mathbf{x}$ from the code.

Siamese networks. A *siamese network* (Bromley et al., 1994; Chopra et al., 2005) applies the *same* network (same weights) to two inputs $\mathbf{x}$ and $\mathbf{x}'$ and compares their embeddings $\mathbf{z} = g_\phi(\mathbf{x})$ and $\mathbf{z}' = g_\phi(\mathbf{x}')$, e.g. with the Euclidean distance. It is trained so that the embeddings of similar pairs are close and those of dissimilar pairs are far apart, for instance with the *contrastive loss* $y \|\mathbf{z} - \mathbf{z}'\|^2 + (1 - y) [\mu - \|\mathbf{z} - \mathbf{z}'\|]_+^2$, where $y = 1$ for a similar pair and $\mu > 0$ is a margin. The network thus learns a metric, which is useful for verification tasks (is this the same person, the same signature?) and when many classes have few examples (face recognition), see Fig. 91.

Generative adversarial networks. A *generative adversarial network* (GAN) (Goodfellow et al., 2014) learns to generate new data similar to the training data. A *generator* G transforms a random noise $\mathbf{z} \sim p_{\mathbf{z}}$ into a fake example $G(\mathbf{z})$, and a *discriminator* D outputs the probability that its input is a real example, see Fig. 92. The two networks play a two-player game:

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [\ln D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [\ln (1 - D(G(\mathbf{z})))] .$$

The discriminator is trained to distinguish the real examples from the fake ones (a binary cross-entropy), and the generator to fool the discriminator. At the equilibrium,

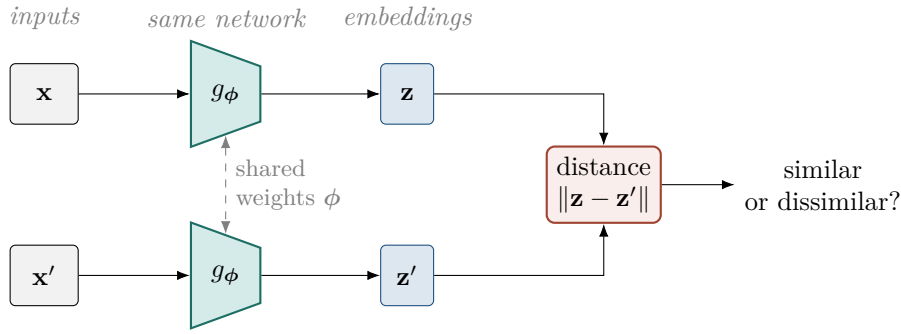


Figure 91: A siamese network: the same network g_ϕ (for images, a CNN) maps each input to an embedding, and the network is trained so that the embeddings of similar inputs are close and those of dissimilar inputs are far apart.

the distribution of $G(\mathbf{z})$ equals the data distribution.

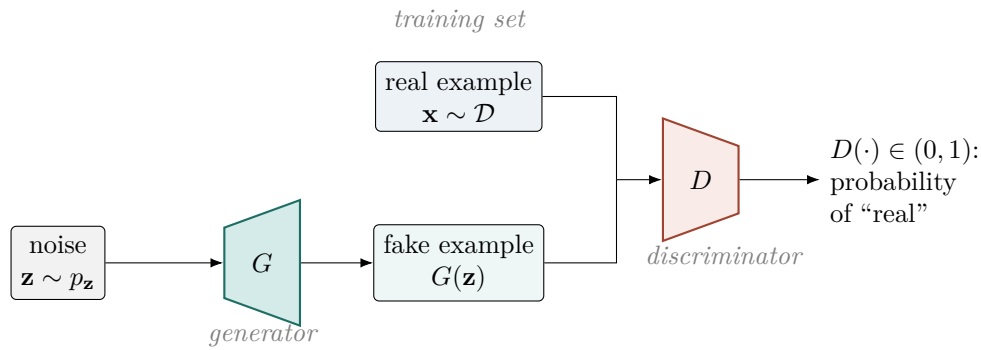


Figure 92: A generative adversarial network: the generator G turns a random noise $\mathbf{z}$ into a fake example $G(\mathbf{z})$, and the discriminator D tries to distinguish the fake examples from the real examples of the training set. D is trained to classify correctly, G to fool D .

Attention and transformers. Recurrent networks process sequences sequentially, which is slow and makes long-range dependencies hard to learn. The *attention* mechanism lets each element of a sequence directly look at all the others. Given queries $\mathbf{Q}$, keys $\mathbf{K}$ and values $\mathbf{V}$ (matrices whose rows are linear projections of the elements of the sequence), it computes

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} \right) \mathbf{V},$$

where the softmax is applied row by row: the output for each element is a weighted average of the values, with weights given by the similarity between its query and the keys. The *transformer* (Vaswani et al., 2017) stacks attention layers, fully connected layers, residual connections and layer normalization, without any recurrence. Its computations are highly parallel, and it has become the standard architecture for natural language processing (large language models are transformers trained to predict the next word), and increasingly for images, sounds and other modalities.

9.11 Practical advice

Training a neural network involves many choices. Here are a few guidelines that work well in most cases.

- **Preprocess the data.** Standardize the input features (zero mean, unit variance); for images, rescale the pixel values. Networks are very sensitive to the scale of the inputs.
- **Start simple.** Begin with a small architecture and a standard configuration (ReLU, He initialization, Adam with $\eta = 10^{-3}$, mini-batches of size 32 to 256), and use a known architecture (or a pre-trained model) for images and texts.
- **Check that the network can overfit.** Before tuning the regularization, make sure that the network reaches a near-zero training loss on a small subset of the data; otherwise, there is a bug or the model is too small.
- **Monitor the training.** Plot the training and validation losses after each epoch. If the training loss does not decrease, decrease the learning rate; if it explodes or becomes NaN, decrease the learning rate or clip the gradients; if the validation loss increases, regularize (early stopping, weight decay, dropout, data augmentation).
- **Tune the learning rate first.** It is the most important hyperparameter; try values on a logarithmic grid ($10^{-4}, \dots, 10^{-1}$) and use a schedule.
- **Use transfer learning.** When few labeled data are available, start from a network pre-trained on a large dataset (*e.g.* ImageNet for images) and fine-tune it on the task at hand.
- **Evaluate properly.** The test set must only be used once, at the end; the hyperparameters (architecture, learning rate, regularization) are selected on a validation set, as for any other model (Section 7).
- **Be reproducible.** Fix the random seeds and keep in mind that the result depends on the initialization: report the mean and standard deviation over several runs.

To remember from Section 9

- A **perceptron** is a linear classifier; its learning rule converges on linearly separable data (Novikoff), but it cannot solve the **XOR** problem.
- A **multilayer perceptron** composes affine maps and non-linear **activation functions** (ReLU in the hidden layers); without non-linearity, it is a linear model. Each layer learns a new representation of the data.
- With one large enough hidden layer, a network can approximate any continuous function (**universal approximation**), but this says nothing about learning and generalization.
- The output activation and the loss go together: linear + squared loss (regression), sigmoid + binary cross-entropy (binary classification, generalizes logistic regression), softmax + cross-entropy (multi-class). In all cases $\delta^{(L)} = f_{\theta}(\mathbf{x}) - \mathbf{y}$.
- **Backpropagation** computes all the gradients in one backward pass through the network, at a cost of the same order as the forward pass:

$$\delta^{(\ell)} = \sigma'_\ell(\mathbf{a}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)\top} \delta^{(\ell+1)}) \text{ and } \partial J / \partial \mathbf{W}^{(\ell)} = \delta^{(\ell)} \mathbf{h}^{(\ell-1)\top}.$$

- Deep networks suffer from **vanishing/exploding gradients**, alleviated by ReLU, Xavier/He initialization, batch normalization and residual connections.
- Training uses **mini-batch SGD** with momentum or **Adam** and a learning-rate schedule; regularization uses weight decay, **early stopping**, dropout and data augmentation.
- **CNNs** (local filters, weight sharing, pooling) are suited to images, **RNNs** (LSTM, GRU) to sequences, and **transformers** (attention) are now the standard for texts.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

From the perceptron to the multilayer perceptron

- Q9.1.** True or false? When the data are linearly separable, the perceptron learning rule returns the separating hyperplane with the largest margin.
- Q9.2.** The inputs satisfy $\|\mathbf{x}_i\| \leq 2$ and are separable with margin $\gamma = 0.1$ (no bias). How many updates can the perceptron make at most?
- Q9.3.** Why can a single perceptron not compute the **XOR** function, and what does the hidden layer of [Fig. 75](#) do?
- Q9.4.** How many parameters does a fully connected network with layer sizes 10–20–20–3 have (biases included)?

Loss functions

- Q9.5.** What does a network with 10 hidden layers and identity activations compute?
- Q9.6.** Compute $\text{softmax}(1, 2, 3)$ and $\text{softmax}(101, 102, 103)$.
- Q9.7.** True or false? Since a network with one hidden layer is a universal approximator, deep networks are useless.
- Q9.8.** A 3-class network outputs $\mathbf{p} = (0.7, 0.2, 0.1)$ on an example of class $y = 2$. What are the loss and its gradient with respect to the scores $\mathbf{z}$?

Regularization

- Q9.9.** A network has $P = 10^6$ parameters. Compare the cost of one gradient computed by backpropagation with that of a finite-difference approximation.
- Q9.10.** Why can the weights of a network not all be initialized to 0?
- Q9.11.** With inverted dropout and a dropout rate $p = 0.2$, by which factor are the kept activations multiplied during training? What is done at test time?
- Q9.12.** What is the standard deviation of the He initialization for a ReLU layer with $n_{\text{in}} = 512$ inputs?

Synthesis of the section

- Q9.13.** How many parameters does a convolution layer with 64 filters of size 3×3 applied to 32 input channels have? Does it depend on the size of the images?

- Q9.14.** A 32×32 image goes through a 5×5 convolution without padding and with stride 1, then through a 2×2 max pooling with stride 2. What is the spatial size of the output?
- Q9.15.** Why does the LSTM suffer less from vanishing gradients than a simple RNN?
- Q9.16.** In a GAN, what does the optimal discriminator output when the generator perfectly reproduces the data distribution?

Part III

Advanced Supervised Machine Learning and Implementation

The algorithms of [Part II](#) learn a single model. This part shows how to go further. [Section 10](#) comes back to statistical learning theory through the decomposition of the error into approximation, estimation and optimization terms, and uses it to explain why *combining* models works: bagging and random forests reduce the variance of unstable learners, boosting and gradient boosting reduce the bias of weak ones, and stacking learns how to combine heterogeneous models. The section ends with an introduction to metric learning, which learns the representation of the data rather than the model. [Section 11](#) gathers practical advice to conduct a machine learning study from the raw data to the final model. The theory of ensemble methods is developed further in [Part IV](#).

10 Advanced Supervised Algorithms

About this section

In this section you will learn to:

- *decompose the error of a predictor into bias, variance and noise, and relate it to the approximation and estimation errors and to the Rademacher complexity;*
- *explain why averaging several models reduces the variance, and implement bagging and random forests (with the out-of-bag error);*
- *run AdaBoost, prove that its training error decreases exponentially fast, and interpret it as a coordinate descent on the exponential loss;*
- *derive gradient boosting as a gradient descent in function space, and understand the second-order criterion of XGBoost;*
- *learn a Mahalanobis distance from labeled data (MMC, ITML, LMNN) to improve distance-based methods.*

10.1 Back to Statistical Learning Theory

In Section 6, we focused on the generalization error and discussed the over-fitting and under-fitting of algorithms by looking at their errors. We have seen that, in practice, we look for the parameters that minimize the error of our algorithm, but we never paid attention to the value of this error itself. More precisely, what is the smallest value our algorithm can possibly reach? If this value is reached, does it mean that the algorithm is perfect? Is it really the best model we could have learned, or even the right class of models?

10.1.1 Error Decomposition

In this part, we will see that the error can be decomposed into several terms, which brings some answers to these questions. This decomposition of the error is also at the origin of more complex models (such as combinations of models), which are designed to reduce some of its components.

The examples of the regression setting presented previously have shown that a simple model, such as a linear one, is sometimes not flexible enough to achieve good performance, *i.e.* it presents a large *bias*. Conversely, a complex model, such as a high-degree polynomial, has a low bias but may not generalize well: it has a high *variance*.

Let us show where these two terms come from by assuming that our data are generated by the following model:

$$y = f(\mathbf{x}) + \varepsilon,$$

where ε is a random noise with zero mean, independent of $\mathbf{x}$, which represents the errors of measurement (or everything that $\mathbf{x}$ cannot explain). Using a sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ generated by this model, we learn a hypothesis h , which is an estimator of f . Because h is learned from the sample S , it is a random quantity. We are now interested in the generalization capacities of h , measured with the mean squared error $\mathbb{E}[(y - h(\mathbf{x}))^2]$ on a new example. More precisely, we study the different components of

this error at a fixed point $\mathbf{x}$, the expectation being taken over the draw of the training sample S and over the noise of the new observation y .

Proposition 10.1: Error Decomposition

We consider the data generation model

$$y = f(\mathbf{x}) + \varepsilon,$$

where $\mathbb{E}[\varepsilon] = 0$. Using a sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ generated by this model, we learn a hypothesis h , which is an estimator of f . Then, for any $\mathbf{x}$, the generalization error of h measured with the mean squared error can be written as:

$$\mathbb{E}[(y - h(\mathbf{x}))^2] = (\mathbb{E}[h(\mathbf{x})] - f(\mathbf{x}))^2 + \mathbb{E}[(\mathbb{E}[h(\mathbf{x})] - h(\mathbf{x}))^2] + \mathbb{E}[(y - f(\mathbf{x}))^2].$$

Proof. We will use the *König-Huygens* formula, which states that for any random variable X with a finite second moment:

$$\mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

We also use the fact that the noise of the new observation y is independent of the training sample S , hence of $h(\mathbf{x})$, and that $\mathbb{E}[y] = f(\mathbf{x})$. We now expand the left-hand side of our equation:

$$\begin{aligned} \mathbb{E}[(y - h(\mathbf{x}))^2] &= \mathbb{E}[y^2 - 2yh(\mathbf{x}) + h(\mathbf{x})^2], \\ &\quad \downarrow \text{linearity of the expectation} \\ &= \underbrace{\mathbb{E}[y^2]} - \mathbb{E}[2yh(\mathbf{x})] + \underbrace{\mathbb{E}[h(\mathbf{x})^2]}, \\ &\quad \downarrow \text{König-Huygens formula and independence of } y \text{ and } h(\mathbf{x}) \\ &= \mathbb{E}[y]^2 + \mathbb{E}[(y - \mathbb{E}[y])^2] - 2\mathbb{E}[y]\mathbb{E}[h(\mathbf{x})] \\ &\quad + \mathbb{E}[h(\mathbf{x})]^2 + \mathbb{E}[(h(\mathbf{x}) - \mathbb{E}[h(\mathbf{x})])^2], \\ &\quad \downarrow \text{using the data generation model, } \mathbb{E}[y] = f(\mathbf{x}) \\ &= f(\mathbf{x})^2 + \mathbb{E}[(y - f(\mathbf{x}))^2] - 2f(\mathbf{x})\mathbb{E}[h(\mathbf{x})] \\ &\quad + \mathbb{E}[h(\mathbf{x})]^2 + \mathbb{E}[(h(\mathbf{x}) - \mathbb{E}[h(\mathbf{x})])^2], \\ &\quad \downarrow \text{factorizing the remaining terms} \\ &= (\mathbb{E}[h(\mathbf{x})] - f(\mathbf{x}))^2 + \mathbb{E}[(\mathbb{E}[h(\mathbf{x})] - h(\mathbf{x}))^2] + \mathbb{E}[(y - f(\mathbf{x}))^2]. \end{aligned}$$

□

In this proposition, the terms on the right-hand side respectively represent:

- the squared **bias** of h : the difference between its mean value over the samples S and the target value $f(\mathbf{x})$;

- the **variance** of h : how much h varies around its mean value when the sample S changes. It represents the sensitivity of the model to the training data;
- the **Bayes error**, which is equal to the variance of the noise ε . It does not depend on the estimator, but only on the distribution of the data: no model can do better.

Example 10.1. Let us consider the two class-conditional densities d_{+1} and d_{-1} represented in Figure 93, with equal class priors. The Bayes classifier predicts, at each point x , the class whose density is the largest. The Bayes error is then related to the area of the region lying under both curves (it is equal to half of this area, here $\mathcal{R}^* \simeq 0.224$), *i.e.* the region where no classifier is able to tell whether a point comes from one distribution or from the other.

As an exercise, you can compute the Bayes error of a two-class problem with equal priors whose class-conditional densities are:

$$d_1(x) = \begin{cases} \frac{3}{2}x^2 + x & \text{when } 0 \leq x \leq 1, \\ 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad d_2(x) = \begin{cases} 1 & \text{when } \frac{3}{4} \leq x \leq \frac{7}{4}, \\ 0 & \text{otherwise.} \end{cases}$$

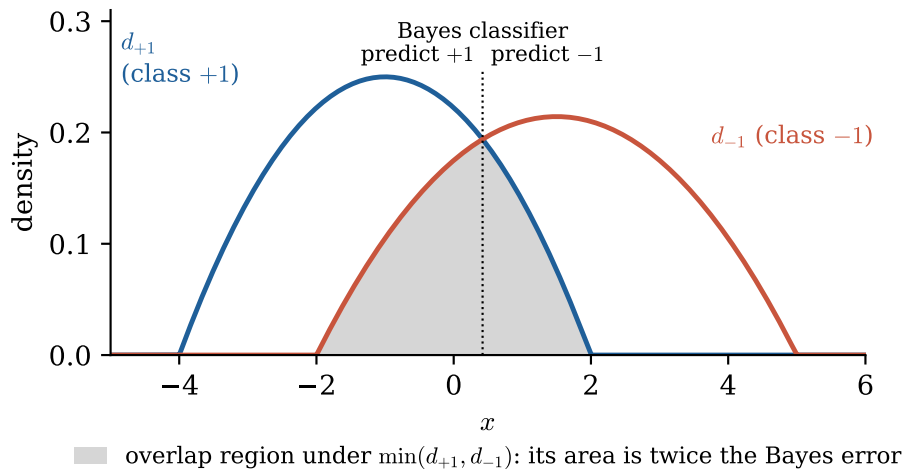


Figure 93: Two class-conditional densities with equal priors, $d_{+1}(x) = (2-x)(x+4)/36$ on $[-4, 2]$ and $d_{-1}(x) = 6(5-x)(x+2)/343$ on $[-2, 5]$. The Bayes classifier predicts the class with the largest density (dotted line). Even this optimal classifier errs on the points of the shaded region, which lies under both curves: the Bayes error is half of its area, $\mathcal{R}^* \simeq 0.224$. [code: `make_figures_ch09.py: fig_bayes_overlap`]

In view of the previous proposition, the quantity we actually aim to minimize is the **excess risk**, that is:

$$\mathbb{E}[(y - h(\mathbf{x}))^2] - \mathbb{E}[(y - f(\mathbf{x}))^2].$$

Although this study has been conducted in the special case of regression with the squared loss, a similar decomposition holds in the general case.

We now suppose, more generally, that the observations are sampled from a joint distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, associated with a probability measure μ . We also consider a loss function ℓ ,

$$\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R},$$

which quantifies the cost incurred by a hypothesis h when it predicts $h(\mathbf{x})$ while the true value is y . The function, or hypothesis, h we are looking for is the one that minimizes the expected error over $\mathcal{D}$, *i.e.* the risk

$$\mathcal{R}(h) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(y, h(\mathbf{x}))] = \int_{\mathcal{X} \times \mathcal{Y}} \ell(y, h(\mathbf{x})) d\mu(\mathbf{x}, y).$$

Keep in mind that, in practice, the hypothesis h is learned from a finite sample. We now assume that we minimize the risk over a function (or hypothesis) space $\mathcal{H}$. If we denote by $\mathcal{R}^*$ the Bayes risk, *i.e.* the smallest risk achievable by any measurable function, we can decompose the **Bayes regret** as:

$$\mathcal{R}(h) - \mathcal{R}^* = \left(\mathcal{R}(h) - \inf_{g \in \mathcal{H}} \mathcal{R}(g) \right) + \inf_{g \in \mathcal{H}} \mathcal{R}(g) - \mathcal{R}^*.$$

The first term is the excess risk of h with respect to the best function of the hypothesis space $\mathcal{H}$, also called the **estimation error**. The second term is the **approximation error**, *i.e.* the smallest excess risk we can achieve with a function of $\mathcal{H}$. This is a **bias** term: it does not depend on the data, but only on the hypothesis space $\mathcal{H}$.

We now draw the link with the generalization bounds presented in Section 6 and go a little further by bounding the **excess risk**. Let us denote by h the hypothesis obtained by minimizing the empirical risk over $\mathcal{H}$ using a sample S :

$$h \in \arg \min_{g \in \mathcal{H}} \mathcal{R}_S(g).$$

We also denote by $h_{\mathcal{H}}$ the minimizer of the risk $\mathcal{R}$ over the hypothesis space $\mathcal{H}$ (we assume that it exists), *i.e.*

$$h_{\mathcal{H}} \in \arg \min_{g \in \mathcal{H}} \mathcal{R}(g).$$

The **excess risk** $\mathcal{R}(h) - \inf_{g \in \mathcal{H}} \mathcal{R}(g)$ can be rewritten as the sum of three terms:

$$\mathcal{R}(h) - \mathcal{R}(h_{\mathcal{H}}) = (\mathcal{R}(h) - \mathcal{R}_S(h)) + (\mathcal{R}_S(h) - \mathcal{R}_S(h_{\mathcal{H}})) + (\mathcal{R}_S(h_{\mathcal{H}}) - \mathcal{R}(h_{\mathcal{H}})),$$

where $(\mathcal{R}(h) - \mathcal{R}_S(h))$ is the difference between the true risk and the empirical risk of the hypothesis h . This is precisely the quantity we are interested in when we study the generalization of algorithms. The second term $(\mathcal{R}_S(h) - \mathcal{R}_S(h_{\mathcal{H}}))$ is non-positive by construction, since h minimizes the empirical risk. The third term $(\mathcal{R}_S(h_{\mathcal{H}}) - \mathcal{R}(h_{\mathcal{H}}))$ is easier to control, as it involves a fixed (deterministic) function to which the law of large numbers applies.

However, the **first term** cannot be handled in the same way, because h depends on the sample S . It can nevertheless be bounded as follows:

$$\mathcal{R}(h) - \mathcal{R}_S(h) \leq \sup_{g \in \mathcal{H}} \left| \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(g(\mathbf{x}), y)] - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right|.$$

Since this quantity also bounds the **third term**, and since the second term is non-positive, we immediately obtain:

$$\mathcal{R}(h) - \mathcal{R}(h_{\mathcal{H}}) \leq 2 \sup_{g \in \mathcal{H}} \left| \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(g(\mathbf{x}), y)] - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right|.$$

This last bound can be seen as a variance term, which increases with the size of $\mathcal{H}$. If $\mathcal{H}$ is large, we can achieve a low empirical risk; however, the true risk is then more likely to be much higher than the empirical one.

A question naturally arises:

How can we define the size of the hypothesis space $\mathcal{H}$?

When $\mathcal{H}$ contains a finite number of hypotheses, the answer is clear: its size is simply the number of hypotheses in the set. But when it is infinite, there exist several measures of the size of $\mathcal{H}$, such as the *Rademacher complexity*.

The Rademacher complexity **Bartlett and Mendelson (2003)** has been introduced to measure the complexity of a set of hypotheses **Koltchinskii and Panchenko (2000)**. Informally, it measures how well the set of hypotheses is able to fit random noise in the dataset.

Definition 10.1: (Empirical) Rademacher complexity

Let $\mathcal{H}$ be a family of functions and $S = \{\mathbf{x}_i\}_{i=1}^m$ a fixed sample of size m . The empirical Rademacher complexity of $\mathcal{H}$ with respect to S is defined as:

$$\mathfrak{R}_S(\mathcal{H}) = \mathbb{E}_{\boldsymbol{\sigma}} \left[\sup_{h \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^m \sigma_i h(\mathbf{x}_i) \right],$$

where $\boldsymbol{\sigma} = (\sigma_1, \dots, \sigma_m)$ is a vector of independent Rademacher random variables, *i.e.* random variables taking the values -1 and $+1$ with probability $1/2$ each.

The Rademacher complexity is the expectation of the above quantity with respect to the distribution $\mathcal{D}$ of the data:

$$\mathfrak{R}_m(\mathcal{H}) = \mathbb{E}_{S \sim \mathcal{D}^m} [\mathfrak{R}_S(\mathcal{H})].$$

This measure increases with the size of $\mathcal{H}$ and, for most classes of interest, decreases with the sample size m .

In a nutshell Draw random labels $\sigma_i \in \{-1, +1\}$ by tossing a coin for each point of S , and look for the hypothesis of $\mathcal{H}$ that agrees best with these meaningless labels. The quantity $\frac{1}{m} \sum_i \sigma_i h(\mathbf{x}_i)$ is the correlation between h and the noise: $\mathfrak{R}_S(\mathcal{H})$ is the best correlation that $\mathcal{H}$ can reach, on average over the coin tosses. A class that contains a single function has a Rademacher complexity equal to 0, whereas a class able to fit *any* labeling of the m points (it *shatters* S) reaches the

maximal value 1 for functions with values in $\{-1, +1\}$. A class which can fit pure noise can also fit the noise present in the data: this is exactly the over-fitting phenomenon.

In the rest of this section, with a slight abuse of notation, we write $\mathfrak{R}_m(\mathcal{H})$ for the Rademacher complexity of the class of losses $\{(\mathbf{x}, y) \mapsto \ell(g(\mathbf{x}), y) : g \in \mathcal{H}\}$ associated with $\mathcal{H}$ (and, in the next few lines, with an absolute value inside the supremum). Using this measure, we will show that

$$\begin{aligned} \mathbb{E}_{S \sim \mathcal{D}^m} [\mathcal{R}(h) - \mathcal{R}(h_{\mathcal{H}})] &\leq 2 \mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(g(\mathbf{x}), y)] - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right| \right], \\ &\leq 4\mathfrak{R}_m(\mathcal{H}). \end{aligned}$$

Therefore,

$$\mathbb{E}_{S \sim \mathcal{D}^m} [\mathcal{R}(h)] - \mathcal{R}^* \leq \inf_{g \in \mathcal{H}} \mathcal{R}(g) - \mathcal{R}^* + 4\mathfrak{R}_m(\mathcal{H}).$$

This result illustrates, in a more general way, the **bias-variance** trade-off in risk minimization: enlarging $\mathcal{H}$ decreases the approximation error but increases the complexity term. It also makes explicit the link between complexity and sample size.

It remains to explain where the following inequality comes from:

$$\mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(g(\mathbf{x}), y)] - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right| \right] \leq 2\mathfrak{R}_m(\mathcal{H}).$$

We prove it using the *symmetrization technique*, which introduces a *ghost sample* $S' = \{(\mathbf{x}'_i, y'_i)\}_{i=1}^m$, drawn from $\mathcal{D}^m$ independently of S . Concretely,

$$\begin{aligned} &\mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} [\ell(g(\mathbf{x}), y)] - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right| \right], \\ &= \mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \mathbb{E}_{S' \sim \mathcal{D}^m} \left[\frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}'_i), y'_i) \right] - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right| \right], \\ &\quad \downarrow \text{the second term does not depend on } S' \\ &= \mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \mathbb{E}_{S' \sim \mathcal{D}^m} \left[\frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}'_i), y'_i) - \frac{1}{m} \sum_{i=1}^m \ell(g(\mathbf{x}_i), y_i) \right] \right| \right], \\ &\quad \downarrow \text{linearity of the empirical mean} \\ &= \mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \mathbb{E}_{S' \sim \mathcal{D}^m} \left[\frac{1}{m} \sum_{i=1}^m (\ell(g(\mathbf{x}'_i), y'_i) - \ell(g(\mathbf{x}_i), y_i)) \right] \right| \right], \\ &\leq \mathbb{E}_{S, S' \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m (\ell(g(\mathbf{x}'_i), y'_i) - \ell(g(\mathbf{x}_i), y_i)) \right| \right], \end{aligned}$$

where the last inequality uses Jensen's inequality ($|\mathbb{E}[X]| \leq \mathbb{E}[|X|]$) and the fact that the supremum of expectations is smaller than the expectation of the supremum.

We now introduce our Rademacher variables σ_i , $i = 1, \dots, m$, and notice that, for any fixed $\boldsymbol{\sigma} \in \{-1, +1\}^m$,

$$\begin{aligned} & \mathbb{E}_{S, S' \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m (\ell(g(\mathbf{x}'_i), y'_i) - \ell(g(\mathbf{x}_i), y_i)) \right| \right] \\ &= \mathbb{E}_{S, S' \sim \mathcal{D}^m} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m \sigma_i (\ell(g(\mathbf{x}'_i), y'_i) - \ell(g(\mathbf{x}_i), y_i)) \right| \right]. \end{aligned}$$

This is due to the fact that the data are *i.i.d.*: multiplying the i -th term by $\sigma_i = -1$ amounts to swapping $(\mathbf{x}_i, y_i)$ and $(\mathbf{x}'_i, y'_i)$, which does not change the distribution of the pair (S, S') . Since this equality holds for any choice of $\boldsymbol{\sigma}$, it also holds when we take the expectation over a vector $\boldsymbol{\sigma}$ of Rademacher variables. Finally, using the triangle inequality,

$$\begin{aligned} & \mathbb{E}_{S, S' \sim \mathcal{D}^m, \boldsymbol{\sigma}} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m \sigma_i (\ell(g(\mathbf{x}'_i), y'_i) - \ell(g(\mathbf{x}_i), y_i)) \right| \right], \\ & \leq \mathbb{E}_{S \sim \mathcal{D}^m, \boldsymbol{\sigma}} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m \sigma_i \ell(g(\mathbf{x}_i), y_i) \right| \right] + \mathbb{E}_{S' \sim \mathcal{D}^m, \boldsymbol{\sigma}} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m \sigma_i \ell(g(\mathbf{x}'_i), y'_i) \right| \right], \\ & = 2 \mathbb{E}_{S \sim \mathcal{D}^m, \boldsymbol{\sigma}} \left[\sup_{g \in \mathcal{H}} \left| \frac{1}{m} \sum_{i=1}^m \sigma_i \ell(g(\mathbf{x}_i), y_i) \right| \right], \\ & = 2\mathfrak{R}_m(\mathcal{H}). \end{aligned}$$

There exist other measures of the capacity of a hypothesis space $\mathcal{H}$, such as the *VC-dimension* [Vapnik and Chervonenkis \(1971\)](#), which is easier to compute when we deal with linear classifiers. We will not study this measure here; instead, we come back to generalization bounds and derive a first bound based on the Rademacher complexity.

10.1.2 Back to Generalization Bounds

Using Definition [10.1](#), we can now state a first generalization bound, given in Theorem [10.2](#).

Theorem 10.2: Rademacher Generalization Bound

Let $\mathcal{H}$ be our class of hypotheses and ℓ a loss function such that $(\mathbf{x}, y) \mapsto \ell(h(\mathbf{x}), y)$ takes its values in $[0, 1]$ for every $h \in \mathcal{H}$. Then, for any $\delta \in (0, 1)$, with probability at least $1 - \delta$ over the draw of an *i.i.d.* sample S of size m , each of the following bounds holds for all $h \in \mathcal{H}$:

$$\mathcal{R}(h) \leq \mathcal{R}_S(h) + 2\mathfrak{R}_m(\mathcal{H}) + \sqrt{\frac{\log(1/\delta)}{2m}}, \quad \text{and}$$

$$\mathcal{R}(h) \leq \mathcal{R}_S(h) + 2\mathfrak{R}_S(\mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2m}},$$

where the Rademacher complexities are those of the class of losses associated with $\mathcal{H}$.

Proof. Let us consider a sample $S = \{\mathbf{z}_i\}_{i=1}^m$, with $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, drawn from $\mathcal{D}^m$, and let us write $\ell(h, \mathbf{z}) = \ell(h(\mathbf{x}), y)$ for short. For any $h \in \mathcal{H}$, we denote by $\mathcal{R}_S(h) = \frac{1}{m} \sum_{i=1}^m \ell(h, \mathbf{z}_i)$ the empirical risk of the hypothesis h .

The proof consists in applying McDiarmid's inequality to the function Φ defined for any sample S by

$$\Phi(S) = \sup_{h \in \mathcal{H}} (\mathcal{R}(h) - \mathcal{R}_S(h)).$$

Let us consider two samples S and S' differing by exactly one point, say $\mathbf{z}_m = (\mathbf{x}_m, y_m) \in S$ and $\mathbf{z}'_m = (\mathbf{x}'_m, y'_m) \in S'$. Since the difference of suprema does not exceed the supremum of the difference, we have

$$\Phi(S) - \Phi(S') \leq \sup_{h \in \mathcal{H}} (\mathcal{R}_{S'}(h) - \mathcal{R}_S(h)) = \sup_{h \in \mathcal{H}} \frac{\ell(h, \mathbf{z}'_m) - \ell(h, \mathbf{z}_m)}{m} \leq \frac{1}{m}.$$

Similarly, we obtain $\Phi(S') - \Phi(S) \leq \frac{1}{m}$, so that $|\Phi(S') - \Phi(S)| \leq \frac{1}{m}$. Then, by McDiarmid's inequality, for any $\delta \in (0, 1)$, with probability at least $1 - \delta$, the following holds:

$$\Phi(S) \leq \mathbb{E}_{S \sim \mathcal{D}^m} [\Phi(S)] + \sqrt{\frac{\log(1/\delta)}{2m}}.$$

Our next step consists in bounding the expectation on the right-hand side of the previous inequality:

$$\begin{aligned} \mathbb{E}_{S \sim \mathcal{D}^m} [\Phi(S)] &= \mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{h \in \mathcal{H}} (\mathcal{R}(h) - \mathcal{R}_S(h)) \right], \\ &= \mathbb{E}_{S \sim \mathcal{D}^m} \left[\sup_{h \in \mathcal{H}} \left(\mathbb{E}_{S' \sim \mathcal{D}^m} [\mathcal{R}_{S'}(h) - \mathcal{R}_S(h)] \right) \right], \\ &\leq \mathbb{E}_{S, S' \sim \mathcal{D}^m} \left[\sup_{h \in \mathcal{H}} (\mathcal{R}_{S'}(h) - \mathcal{R}_S(h)) \right], \\ &= \mathbb{E}_{S, S' \sim \mathcal{D}^m} \left[\sup_{h \in \mathcal{H}} \left(\frac{1}{m} \sum_{i=1}^m (\ell(h, \mathbf{z}'_i) - \ell(h, \mathbf{z}_i)) \right) \right], \\ &= \mathbb{E}_{S, S' \sim \mathcal{D}^m, \sigma} \left[\sup_{h \in \mathcal{H}} \left(\frac{1}{m} \sum_{i=1}^m \sigma_i (\ell(h, \mathbf{z}'_i) - \ell(h, \mathbf{z}_i)) \right) \right], \\ &\leq \mathbb{E}_{S' \sim \mathcal{D}^m, \sigma} \left[\sup_{h \in \mathcal{H}} \left(\frac{1}{m} \sum_{i=1}^m \sigma_i \ell(h, \mathbf{z}'_i) \right) \right] \\ &\quad + \mathbb{E}_{S \sim \mathcal{D}^m, \sigma} \left[\sup_{h \in \mathcal{H}} \left(\frac{1}{m} \sum_{i=1}^m (-\sigma_i) \ell(h, \mathbf{z}_i) \right) \right], \end{aligned}$$

$$\begin{aligned}
& \downarrow \sigma \text{ and } -\sigma \text{ have the same distribution} \\
& = 2 \mathbb{E}_{S \sim \mathcal{D}^m, \sigma} \left[\sup_{h \in \mathcal{H}} \left(\frac{1}{m} \sum_{i=1}^m \sigma_i \ell(h, \mathbf{z}_i) \right) \right], \\
& = 2\mathfrak{R}_m(\mathcal{H}).
\end{aligned}$$

This proves the first bound. The reader has probably noticed that this step of the proof is exactly the same as the one used when studying the excess risk.

For the second bound, we note that changing one point of S changes $\mathfrak{R}_S(\mathcal{H})$ by at most $1/m$. Using McDiarmid's inequality again, we get, with probability at least $1 - \delta/2$,

$$\mathfrak{R}_m(\mathcal{H}) \leq \mathfrak{R}_S(\mathcal{H}) + \sqrt{\frac{\log(2/\delta)}{2m}}.$$

Applying the first bound with $\delta/2$ instead of δ and combining the two inequalities with a union bound, we obtain that, with probability at least $1 - \delta$:

$$\Phi(S) \leq 2\mathfrak{R}_S(\mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2m}}.$$

□

10.2 Ensemble Methods

In what we have seen so far, we only used one model to answer a given problem, *e.g.* a linear regression to predict the price of a house given its characteristics, or a logistic regression to predict whether someone is infected or not.

But why use only one model? Would it not be more interesting to use several models to build a more powerful one?

The answer is indeed yes: it is often more interesting to do so, and combinations of models are currently among the best-performing methods on many tasks.

But how can we create such a combination of models?

A naive rule Let us imagine that we want to create T models using a sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ of size m . We can train a hypothesis h_t , for t from 1 to T , using the same training sample S for each t .

Then, we can combine the different hypotheses into a single one, denoted by H_T , by averaging them to take our final decision:

- for a regression task, the predicted value is the mean of the predictions made by the hypotheses h_t , *i.e.*

$$H_T(\mathbf{x}) = \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}).$$

- for a (binary) classification task, we can apply the same rule. However, instead of taking the mean value as the output of the combined classifier, we take the sign of this mean value, *i.e.*

$$H_T(\mathbf{x}) = \text{sign} \left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right).$$

When the hypotheses h_t return a value in $\{-1, 1\}$, the hypothesis H_T can then be seen as a majority vote with equal weights. On the other hand, if the hypotheses h_t return real values, we can interpret H_T as a weighted majority vote, where the weight of each vote is given by the confidence $|h_t(\mathbf{x})|$ of the hypothesis.

Although this rule is simple, it is not very useful in practice. Indeed, if the learning problems we solve are convex, the hypotheses h_t are very likely to be (almost) identical when we use the same training set each time.

Varying the hyper-parameters One could then be tempted to diversify the hypotheses by imposing different hyper-parameter values, *e.g.* one could choose T different hyper-parameter values in order to obtain T different models.

Again, this solution is not satisfactory. It may even lead to hypotheses with a low predictive power. Moreover, such a process would conflict with the cross-validation procedure presented earlier, which is precisely used to optimize the values of these hyper-parameters.

We will see that, even if our algorithm does not depend on hyper-parameters, it is possible to improve it using a single training set S . The two main methods to do so are known as **Bagging** and **Boosting**. They act differently on the performance of algorithms, or more precisely on the different components of their error. We will also present a third, more general, method, which consists in learning how to combine several algorithms. But before presenting these approaches, let us motivate the use of several models.

A theoretical analysis We now explain why it is interesting to combine several models. To do so, let us consider data $(\mathbf{x}, y)$ drawn from a distribution $\mathcal{D}$, where $\mathbf{x}$ is the feature vector and y is the response variable, *i.e.* the value to predict.

In a regression setting, we learn a hypothesis h which aims to predict the value y from $\mathbf{x}$. For the sake of simplicity, we suppose that there exists a *true function* r such that $r(\mathbf{x}) = y$ for all $(\mathbf{x}, y) \sim \mathcal{D}$. We also consider a training set S of size m and recall that we aim to learn a set of hypotheses h_t , $t = 1, \dots, T$. Thus, for any instance $\mathbf{x}$ and any hypothesis h_t ,

$$h_t(\mathbf{x}) = r(\mathbf{x}) + \varepsilon_t(\mathbf{x}),$$

where $\varepsilon_t(\mathbf{x})$ is the error between the value predicted by h_t and the true value of the function at $\mathbf{x}$.

Keep in mind that, in a regression setting, the error we consider is the MSE, *i.e.* the mean squared error. Thus, the generalization error of a single hypothesis h_t is defined by:

$$\mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[(h_t(\mathbf{x}) - r(\mathbf{x}))^2 \right] = \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\varepsilon_t(\mathbf{x})^2 \right],$$

where $\mathcal{D}_{\mathcal{X}}$ denotes the marginal distribution of $\mathbf{x}$. The generalization error of our averaged hypothesis $H_T = \frac{1}{T} \sum_{t=1}^T h_t$ is

$$\begin{aligned} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[(H_T(\mathbf{x}) - r(\mathbf{x}))^2 \right] &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) - r(\mathbf{x}) \right)^2 \right], \\ &\quad \downarrow \text{applying the definition of } \varepsilon_t \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\left(\frac{1}{T} \sum_{t=1}^T \varepsilon_t(\mathbf{x}) \right)^2 \right]. \end{aligned}$$

Let us now suppose that the errors ε_t are centered (with a mean value equal to 0) and uncorrelated. We can rewrite the previous expression as:

$$\begin{aligned} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[(H_T(\mathbf{x}) - r(\mathbf{x}))^2 \right] &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\left(\frac{1}{T} \sum_{t=1}^T \varepsilon_t(\mathbf{x}) \right)^2 \right], \\ &\quad \downarrow \text{we expand the square} \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\left(\frac{1}{T} \sum_{t_1=1}^T \varepsilon_{t_1}(\mathbf{x}) \right) \left(\frac{1}{T} \sum_{t_2=1}^T \varepsilon_{t_2}(\mathbf{x}) \right) \right], \\ &\quad \downarrow \text{we use the fact that } \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} [\varepsilon_{t_1}(\mathbf{x}) \varepsilon_{t_2}(\mathbf{x})] = 0 \text{ for all } t_1 \neq t_2 \\ &= \frac{1}{T^2} \sum_{t=1}^T \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\varepsilon_t(\mathbf{x})^2 \right], \\ &= \frac{1}{T} \left(\frac{1}{T} \sum_{t=1}^T \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\varepsilon_t(\mathbf{x})^2 \right] \right). \end{aligned}$$

This last equation shows that the generalization error of the combination of hypotheses is equal to the average error of the T hypotheses divided by T .

Note that, in practice, the assumption of uncorrelated errors is essentially wrong, because the hypotheses h_t are learned using similar samples S_t . However, we can still show that the error of the averaged hypothesis is never larger than the average error of the hypotheses h_t . Indeed, the function $u \mapsto u^2$ is convex, so Jensen's inequality gives, for every $\mathbf{x}$,

$$\left(\frac{1}{T} \sum_{t=1}^T \varepsilon_t(\mathbf{x}) \right)^2 \leq \frac{1}{T} \sum_{t=1}^T \varepsilon_t(\mathbf{x})^2.$$

Taking the expectation over $\mathbf{x}$, we immediately obtain:

$$\mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[(H_T(\mathbf{x}) - r(\mathbf{x}))^2 \right] \leq \frac{1}{T} \sum_{t=1}^T \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}} \left[\varepsilon_t(\mathbf{x})^2 \right].$$

Let us now go on with a first ensemble approach: the *bagging* procedure.

10.2.1 Bagging and Random Forests

Bagging Bagging [Breiman \(1996\)](#) is a way to combine models that perform well on the training set, *i.e.* models with a low bias. We have previously seen that we can bound the expected *Bayes regret* by the sum of two terms:

$$\mathbb{E}_{S \sim \mathcal{D}^m} [\mathcal{R}(h)] - \mathcal{R}^* \leq \inf_{g \in \mathcal{H}} \mathcal{R}(g) - \mathcal{R}^* + 4\mathfrak{R}_m(\mathcal{H}).$$

Here, we are more precisely interested in the variance term of the error decomposition. This term tells us how sensitive the algorithm is to variations of the training set: if its value is low, our algorithm is only slightly affected by a change of the training set.

Let us illustrate this with a small example, depicted in [Figure 94](#), for a regression task (the situation is similar for classification tasks). We learn a polynomial regression model of degree 15 on different datasets drawn from the same distribution, each training set being of size 30. In total, we learn 10 different models, which are represented on the left-hand graph. We note that the variance of the models is very large, *i.e.* for the same value on the x -axis, the models return very different values on the y -axis: the models are very sensitive to the training data.

On the other hand, if we average the predictions of the 10 models, we obtain the right-hand graph. We notice that the combined model varies much less around the true regression function. We have therefore succeeded in reducing the variance by combining several models.

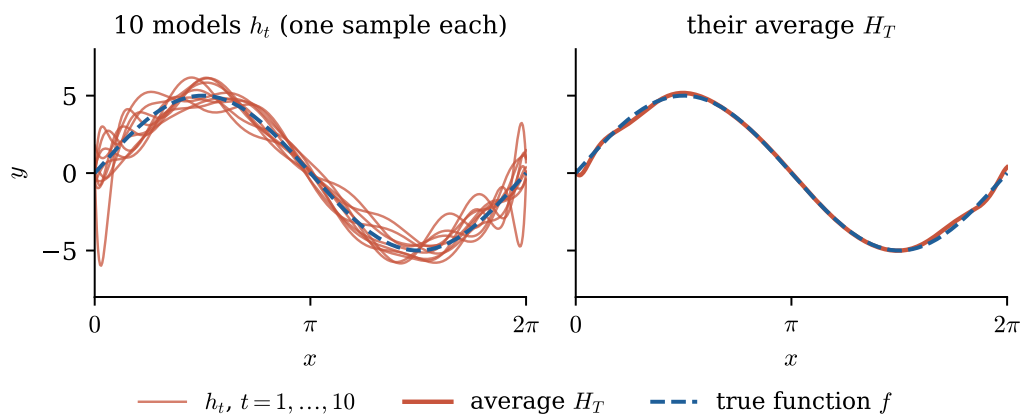


Figure 94: Illustration of the variance reduction obtained by combining models (polynomials of degree 15, regression function $f(x) = 5 \sin(x)$, Gaussian noise of variance 1). Left: the 10 models h_t learned on 10 different datasets of size 30 drawn from the same distribution (red), together with the regression function (dashed blue); right: the model H_T obtained by averaging the 10 previous models.

[code: `make_figures_ch09.py`: `fig_bagging_variance`]

With bagging, we wish to exploit the fact that the variance of a combination of models is smaller than the variance of the individual models. Moreover, the combined model retains a strong predictive power, *i.e.* it remains accurate.

In this example, the models are learned on different datasets. But, in practice, we only have one training set S . We therefore have to find a way to create several training sets S_t from this set S . This can be done using the *bootstrap* [Efron \(1979\)](#), a resampling method based on random draws. How does it work in practice?

Algorithm 6: Bagging

Inputs : Training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, number of models T

Output: Hypothesis H_T

- 1 **for** $t = 1$ **to** T **do**
 - 2 create a bootstrap sample S_t of size m from S ;
 - 3 learn a hypothesis h_t using S_t ;
 - 4 set $H_T = \frac{1}{T} \sum_{t=1}^T h_t$ (or its sign, or a majority vote, for classification);
-

Let us consider a training set S of size m . To create a bootstrap sample S_t of the same size m , we simply draw m examples from S uniformly at random and with replacement, *i.e.* each example of S has the same probability $1/m$ of being drawn at each draw.

With this procedure, the same example may occur several times in the same training set. Thus, in order to minimize its error, the algorithm has to pay more attention to such an example and gives it more importance. This sampling method therefore creates diversity among the learned models. Indeed, since the sets S_t are different, the hypotheses h_t focus on different regions of the data space. Conversely, some examples are never drawn: they are said to be *out-of-bag* for the hypothesis h_t , and we will see below that they can be used to estimate its error for free (see Figure 95).

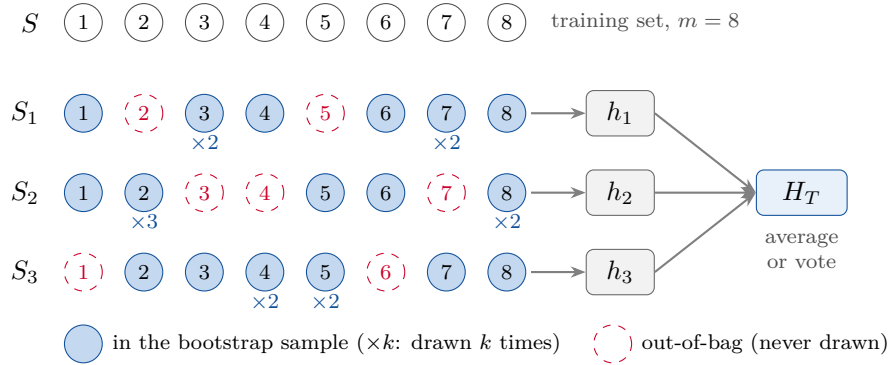


Figure 95: Bagging with $T = 3$ bootstrap samples of size $m = 8$. Each S_t is obtained by 8 draws with replacement from S : some examples are drawn several times, others are never drawn (dashed red circles) and are *out-of-bag* for h_t . For instance, example 2 is out-of-bag for h_1 only, so that its OOB prediction is made by h_1 alone, while the hypotheses h_2 and h_3 , which have seen it, are not used.

The bagging procedure is summarized in Algorithm 6. As the reader may guess, *bagging* stands for *bootstrap aggregating*: we generate several samples and hypotheses, and then aggregate their predictions.

Before presenting one of the most famous models based on this process, *the random forest algorithm*, let us see from a theoretical point of view why it is interesting to combine these models.

Theoretical analysis of bagging We can conduct an analysis similar to the one presented in the previous section. Let us denote by $H_T = \frac{1}{T} \sum_{t=1}^T h_t$ our bagging

hypothesis, used to estimate (for instance) the output y of a regression function, *i.e.* $H_T(\mathbf{x}) \simeq y$. Then, the average error made by the set of hypotheses is equal to:

$$\mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[\frac{1}{T} \sum_{t=1}^T (h_t(\mathbf{x}) - y)^2 \right] = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[y^2 - 2y \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) + \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x})^2 \right].$$

Using Jensen's inequality again, we have $\left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right)^2 \leq \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x})^2$, so that

$$\begin{aligned} \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[\frac{1}{T} \sum_{t=1}^T (h_t(\mathbf{x}) - y)^2 \right] &\geq \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[y^2 - 2y \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) + \left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right)^2 \right], \\ &= \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[\left(y - \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right)^2 \right], \\ &= \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[(y - H_T(\mathbf{x}))^2 \right]. \end{aligned}$$

This last inequality shows that the error of the bagging hypothesis is always lower than or equal to the average error of the individual hypotheses.

We can go a little further by studying the potential gain of the bagging hypothesis. This gain is given by the expectation of the difference between the two sides of the inequality

$$\left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right)^2 \leq \frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x})^2,$$

that is,

$$\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x})^2 - \left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right)^2 = \frac{1}{T} \sum_{t=1}^T (h_t(\mathbf{x}) - H_T(\mathbf{x}))^2,$$

which is the variance (or *ambiguity* [Krogh and Vedelsby \(1995\)](#)) of the predictions of our set of hypotheses h_t at the point $\mathbf{x}$. If this variance is large, then the gain is large. This shows that, to obtain a good bagging hypothesis, it is more interesting to use base hypotheses with a high variance but a low bias, which is exactly the case of **decision trees**. This is why it is interesting to combine them using this procedure, which leads us to the **random forest** algorithm.¹⁷

Random forests When we introduced decision trees, we said that the size of a tree, *i.e.* its depth, depends on the data: the tree grows until we obtain pure leaves.

If we go back to our error decomposition, (deep) decision trees are thus hypotheses with a low bias (their training error decreases with the depth) but with a high variance:

¹⁷Note that a similar study can be conducted in the case of classification, but it is not presented here.

Algorithm 7: Random Forests

Inputs : Training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, number of trees T , a sample size m' and a number of features p'

Output: Hypothesis H_T

- 1 **for** $t = 1$ **to** T **do**
 - 2 create a bootstrap sample S_t of size $m' \leq m$ from S ;
 - 3 build a decision tree h_t on S_t where, at each node, only a random subset of p' features is considered to split the node;
 - 4 **set** $H_T = \frac{1}{T} \sum_{t=1}^T h_t$ (majority vote of the trees for classification);
-

they are very sensitive to the data, and their structure can vary greatly from one training set to another.

In fact, we are in exactly the same situation as in the regression example presented in Figure 94. We therefore combine tree models using bagging, in order to reduce the variance of the trees while preserving their predictive capacities. This combination of trees by bagging gives rise to the random forest algorithm, introduced by Breiman at the beginning of the 21st century [Breiman \(2001\)](#).

The idea is to build several trees on different training sets S_t drawn randomly with replacement from S , *i.e.* using the bootstrap procedure, and to combine the predictions of the different trees. However, the random forest algorithm (presented in Algorithm 7) is in fact more sophisticated than that: it is based on the principle of double sampling, *i.e.* sampling both the examples and the features.

This double sampling creates diversity both at the level of the examples and at the level of the features. Compared to the standard bagging algorithm, note that the bootstrap sample has a size $m' \leq m$ and that the number p' of features considered at each node of a tree is smaller than the dimension d of the data (a common default choice is $p' \simeq \sqrt{d}$ for classification). Note that it is not mandatory to have $m' < m$: it only makes the learning procedure faster. From an algorithmic point of view, both kinds of sampling speed up the learning of the trees: there are fewer examples, and the splitting procedure is applied to fewer features.

In a nutshell: why sample the features? Our analysis of the averaged hypothesis assumed uncorrelated errors, which is far too optimistic for trees grown on bootstrap samples of the same dataset: if a feature is very informative, all the trees split on it first and look alike. If the T hypotheses have the same variance σ^2 at $\mathbf{x}$ and a pairwise correlation $\rho \geq 0$, a direct computation gives ([Hastie et al., 2009](#), Chapter 15)

$$\text{Var} \left(\frac{1}{T} \sum_{t=1}^T h_t(\mathbf{x}) \right) = \rho \sigma^2 + \frac{1-\rho}{T} \sigma^2.$$

Adding trees only kills the second term: the first one, $\rho \sigma^2$, remains whatever the size of the forest. Drawing a random subset of p' features at each node decorrelates the trees (it decreases ρ), at the price of slightly worse individual trees (a small increase of the bias and of σ^2): this is the whole idea of random forests compared with plain bagging.

The presented algorithm is the simplest version. It is not rare to give different weights to the trees according to their performance: remember, this is our weighted majority vote.

Decision trees and random forests are used in many applications, such as finance, security or the social sciences in general. These algorithms have the advantage of being easy to build. Moreover, the decision of a single tree is easy to understand (explainable AI): you just have to follow the path of the example in the tree. A forest of hundreds of trees is less transparent, but tools such as the feature importance presented below help to interpret it.

Out-of-bag (OOB) error for validation The bootstrap procedure implies that we do not use all the data to learn each hypothesis. More precisely, given a sample S of size m , if every example of S is drawn with the same probability $1/m$, then the probability of never selecting a given example in m draws is equal to $(1 - 1/m)^m$. When the sample size m is large enough, this quantity tends towards $e^{-1} \simeq 0.37$.

It means that the bagging procedure does not use approximately 37% of the (distinct) examples when learning a base hypothesis: only approximately 63% of them are used. We can wonder how these remaining data can be exploited in the bagging procedure. These examples are called **out-of-bag (OOB) data**. Each example $\mathbf{x}_i$ is out-of-bag for roughly a third of the base hypotheses; we can therefore predict its label using only these hypotheses and compare this prediction with y_i . Averaging these errors over the training set gives the **OOB error**, an almost free estimate of the generalization error, which can be used to tune the hyper-parameters of the model without a separate validation set.

Bagging is thus mainly interesting to

- obtain an ensemble with a lower variance than its base hypotheses;
- train a model with better generalization capacities at a moderate cost, since the base models can be learned in parallel and the OOB error provides a validation for free.

Figure 96 illustrates these properties on a simulated classification problem: a single deep tree over-fits, bagging and random forests quickly improve on it as trees are added, and their test error then reaches a plateau. In particular, adding trees does not lead to over-fitting: increasing T only reduces the Monte Carlo variability of the ensemble, so that T is not a hyper-parameter that needs careful tuning (it is enough to take it large enough).

Pitfall: bagging is not a universal booster Bagging only reduces the [variance](#). Bagging a *stable* learner with a high bias, such as a linear regression or a k -NN classifier with a large k , hardly changes its predictions: all the bootstrap models are almost identical, and the bias is unchanged. Bagging pays off for unstable, low-bias learners such as deep trees or flexible neural networks [Breiman \(1996\)](#). Note also that the OOB error is only a fair estimate of the generalization error when the examples are *i.i.d.*: with repeated measurements of the same individual, or with time series, an example and its “copies” may fall on both sides of the bootstrap split, exactly as for the cross-validation procedure (Section 11).

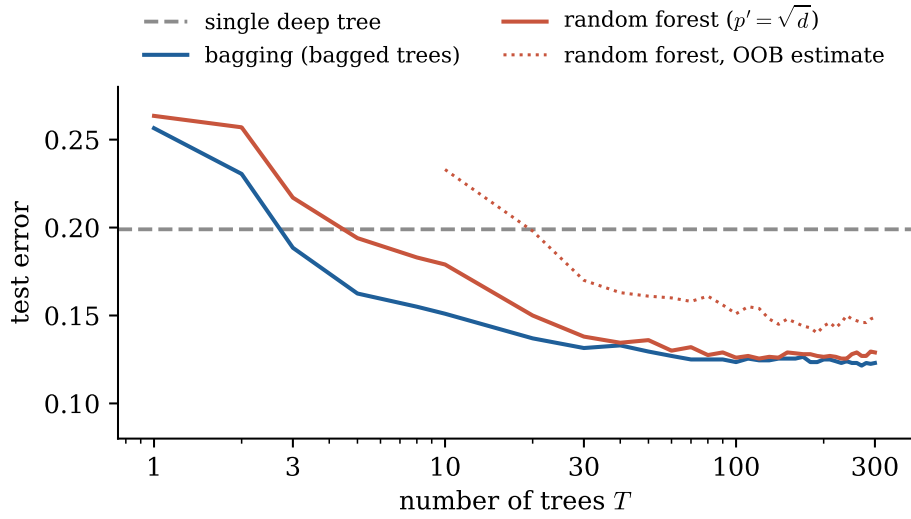


Figure 96: Test error of a single fully grown tree, of bagged trees and of a random forest as a function of the number of trees T (logarithmic scale), on a simulated problem with $d = 20$ features and 1000 training examples. The dotted line is the OOB estimate of the error of the forest: it is computed for free from the training set and follows the test error, slightly pessimistically, since each OOB prediction only uses about a third of the trees. [code: `make_figures_ch09.py: fig_forest_ntrees`]

As for simple decision trees, random forests can be used for both regression and classification: the main difference lies in the way the trees are built, *i.e.* in the impurity measure (or loss function) used.

Measuring the importance of variables As for the decision trees presented in Section 8.7, we can measure the importance of each feature X^j in the set of trees. This way of measuring feature importance was introduced together with the random forest algorithm Breiman (2001).

Remember that, to assess the quality of a split of a node N according to a feature X^j in a decision tree, we measure the decrease of entropy $\Delta Ent_N(X^j)$ between before and after the split (we present it for binary decision trees for the sake of simplicity):

$$\Delta Ent_N(X^j) = Ent(\mathbf{p}_N) - Ent(\mathbf{p}_N, X^j),$$

where $Ent(\mathbf{p}_N) = -\sum_{k=1}^C p_{N,k} \log_2(p_{N,k})$ and

$$Ent(\mathbf{p}_N, X^j) = -\frac{|N_L|}{|N|} \sum_{k=1}^C p_{N_L,k} \log_2(p_{N_L,k}) - \frac{|N_R|}{|N|} \sum_{k=1}^C p_{N_R,k} \log_2(p_{N_R,k}).$$

In these expressions, N_L and N_R denote the two children (left and right) obtained after the split of the node N , $|N|$ denotes the number of training examples reaching the node N , and $p_{N,k}$ is the proportion of examples of class k in the node N .

To measure the importance Imp^j of the variable X^j , we sum the decreases $\Delta Ent_N(X^j)$ over the nodes N that are split using X^j , each decrease being weighted by the proportion of training examples reaching the node, and we average over the trees:

$$Imp^j = \frac{1}{T} \sum_{t=1}^T \sum_{N \in \mathcal{N}_t} \frac{|N|}{m_t} \Delta Ent_N(X^j) \mathbb{1}_{\{X^j \text{ is used to split } N\}},$$

where $\mathcal{N}_t$ denotes the set of internal nodes of the t -th tree, m_t the size of the training sample of the t -th tree, and T the total number of trees. The entropy can of course be replaced by any other impurity measure, such as the Gini index: this measure is known as the *mean decrease in impurity*. Note that it is computed on the training data and tends to favor features with many possible split values; the *permutation importance*, also introduced in Breiman (2001) and discussed in Section 11, is a useful complement.

We will see in the next section that there is another way to combine trees (or other classifiers or regressors) of limited capacity, called *weak hypotheses*: the *boosting* procedure.

10.2.2 Boosting

This section is largely inspired by Chapter 7 of the book *Foundations of Machine Learning* Mohri et al. (2012).

Setting Let us first introduce the definitions of *strong* and *weak* learnability. They are given in the context of classification, where boosting is mainly used, so that we can use the terms classifier and hypothesis interchangeably.

Definition 10.2: Strong Learnability Mohri et al. (2012)

A concept class $\mathcal{C}$ is said to be (strongly) PAC-learnable if there exist an algorithm $\mathcal{A}$ and a polynomial function *poly* such that for any $\varepsilon > 0$ and $\delta > 0$, for all distributions $\mathcal{D}$ over $\mathcal{X}$ and for any target concept $c \in \mathcal{C}$, the following holds for any sample size $m \geq \text{poly}(1/\varepsilon, 1/\delta, d, \text{size}(c))$:

$$\mathbb{P}_{S \sim \mathcal{D}^m} [\mathcal{R}(h_S) \leq \varepsilon] \geq 1 - \delta,$$

where h_S is the hypothesis returned by $\mathcal{A}$ when trained on S . If $\mathcal{A}$ further runs in $\text{poly}(1/\varepsilon, 1/\delta, d, \text{size}(c))$, then $\mathcal{C}$ is said to be efficiently PAC-learnable.

In this definition, a concept c is a function from $\mathcal{X}$ to $\mathcal{Y}$ that we want to learn. For example, a concept may be the indicator function of the set of points lying inside a given triangle. A concept class, denoted by $\mathcal{C}$, is a set of concepts we may wish to learn, for example the set of (indicator functions of) all triangles in the plane.

According to the previous definition, a concept class $\mathcal{C}$ is PAC-learnable if the hypothesis returned by the algorithm after observing a number of points polynomial in $1/\varepsilon$ and $1/\delta$ is approximately correct (its error is at most ε) with high probability (at least $1 - \delta$).

Definition 10.3: Weak Learnability Mohri et al. (2012)

A concept class $\mathcal{C}$ is said to be weakly PAC-learnable if there exist an algorithm $\mathcal{A}$, a constant $\gamma > 0$ and a polynomial function poly such that for any $\delta > 0$, for all distributions $\mathcal{D}$ over $\mathcal{X}$ and for any target concept $c \in \mathcal{C}$, the following holds for any sample size $m \geq \text{poly}(1/\delta, d, \text{size}(c))$:

$$\mathbb{P}_{S \sim \mathcal{D}^m} \left[\mathcal{R}(h_S) \leq \frac{1}{2} - \gamma \right] \geq 1 - \delta,$$

where h_S is the hypothesis returned by $\mathcal{A}$ when trained on S . When such an algorithm exists, it is called a weak learning algorithm, a weak learner or a base classifier.

The idea is quite similar to that of strong learners. The difference between the two definitions lies in the required error of the hypothesis h_S . In the first one, we require the classifier to achieve an error of at most ε , for any $\varepsilon > 0$, given a sufficiently large sample. In the second one, we just want the classifier to be slightly better (by a margin $\gamma > 0$) than random guessing.

We have already met strong classifiers, such as deep decision trees or over-parameterized neural networks. Weak classifiers are also naturally present in the literature, *e.g.* small decision trees with a depth of one or two (decision stumps), or linear classifiers such as a linear SVM when the problem is not linearly separable.

Whereas bagging combines strong classifiers (with a low bias and a high variance), boosting combines weak ones. It tries to combine them in order to build a strong classifier, but in a way that is different from the bagging procedure. Instead of building bootstrap samples, boosting modifies the distribution of the data in order to build a sequence of classifiers $(h_t)_{t \in \mathbb{N}}$, where h_{t+1} tries to correct the errors made by the previous classifiers.

Historical note Boosting was born from a purely theoretical question raised by Kearns and Valiant at the end of the 1980s [Kearns and Valiant \(1994\)](#): are weak and strong learnability equivalent? In other words, can a learner that is only slightly better than random guessing always be “boosted” into an arbitrarily accurate one? Schapire answered positively in 1990 with a first, rather impractical, recursive construction [Schapire \(1990\)](#). A few years later, Freund and Schapire proposed AdaBoost [Freund and Schapire \(1997\)](#), simple and adaptive (it does not need to know the edge γ in advance), which turned out to be remarkably efficient in practice; they received the Gödel Prize in 2003 for this work. The statistical interpretation of boosting as the minimization of a loss by a greedy, stage-wise procedure [Friedman et al. \(2000\)](#); [Friedman \(2001\)](#) then opened the way to gradient boosting (Section 10.3). A complete account is given in the book [Schapire and Freund \(2012\)](#).

AdaBoost A well-known boosting algorithm is *AdaBoost* [Freund and Schapire \(1999\)](#), which iteratively focuses on the examples that are difficult to classify. The algorithm is presented in Algorithm 8, and an illustration is provided in Figure 97. Here, the labels are in $\{-1, +1\}$ and the base classifiers h_t take their values in $\{-1, +1\}$.

Algorithm 8: AdaBoost

Inputs : Training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ with $y_i \in \{-1, +1\}$, number of rounds T
Output: Hypothesis H_T

```
1 for  $i = 1$  to  $m$  do
2    $\lfloor$  set the weight of the  $i$ -th example to  $w_i^{(1)} = 1/m$ ;
3 for  $t = 1$  to  $T$  do
4   learn a base classifier  $h_t$  using  $S$  weighted by  $\mathbf{w}^{(t)}$ ;
5   compute the weighted error  $\varepsilon_t = \sum_{i=1}^m w_i^{(t)} \mathbb{1}_{\{h_t(\mathbf{x}_i)y_i < 0\}}$  of  $h_t$ ; // we assume
    $\varepsilon_t < 1/2$ 
6   set  $\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \varepsilon_t}{\varepsilon_t} \right)$ ;
7   set  $Z_t = \sum_{i=1}^m w_i^{(t)} \exp(-\alpha_t y_i h_t(\mathbf{x}_i)) = 2\sqrt{\varepsilon_t(1 - \varepsilon_t)}$ ;
8   for  $i = 1$  to  $m$  do
9      $\lfloor$  set  $w_i^{(t+1)} = w_i^{(t)} \frac{\exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{Z_t}$ ;
10 set  $f_T = \sum_{t=1}^T \alpha_t h_t$  and  $H_T = \text{sign}(f_T)$ ;
```

Let us say a few words about this algorithm before analyzing it.

At round t , the i -th training example has the weight $w_i^{(t)}$. For $t = 1$, all the training examples have the same weight, equal to $1/m$. A hypothesis h_t is learned, and we compute its weighted classification error ε_t :

$$\varepsilon_t = \sum_{i=1}^m w_i^{(t)} \mathbb{1}_{\{h_t(\mathbf{x}_i)y_i < 0\}}.$$

Using this value, we compute the *weight* α_t of the learned classifier:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right).$$

Keep in mind that the idea is to learn a hypothesis H_T which is the sign of a linear combination of the weak learners h_t . The better the weak learner (the smaller ε_t), the larger its weight; note that $\alpha_t > 0$ as soon as $\varepsilon_t < 1/2$.

The rest of the procedure consists in finding a good reweighting of the training examples such that, during the next round, the new classifier h_{t+1} is able to correct the mistakes made by the classifier h_t . This is done using the following update rule:

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{Z_t},$$

where Z_t is a normalization factor which ensures that the weights sum to 1. We will show (see the proof below) that $Z_t = 2\sqrt{\varepsilon_t(1 - \varepsilon_t)}$. Since $\alpha_t > 0$, the weights of the misclassified examples ($y_i h_t(\mathbf{x}_i) = -1$) are multiplied by e^{α_t}/Z_t , whereas those of the correctly classified examples are multiplied by $e^{-\alpha_t}/Z_t$. This reweighting thus gives more weight to the examples misclassified by h_t and, conversely, decreases the weight of the examples that are correctly classified.

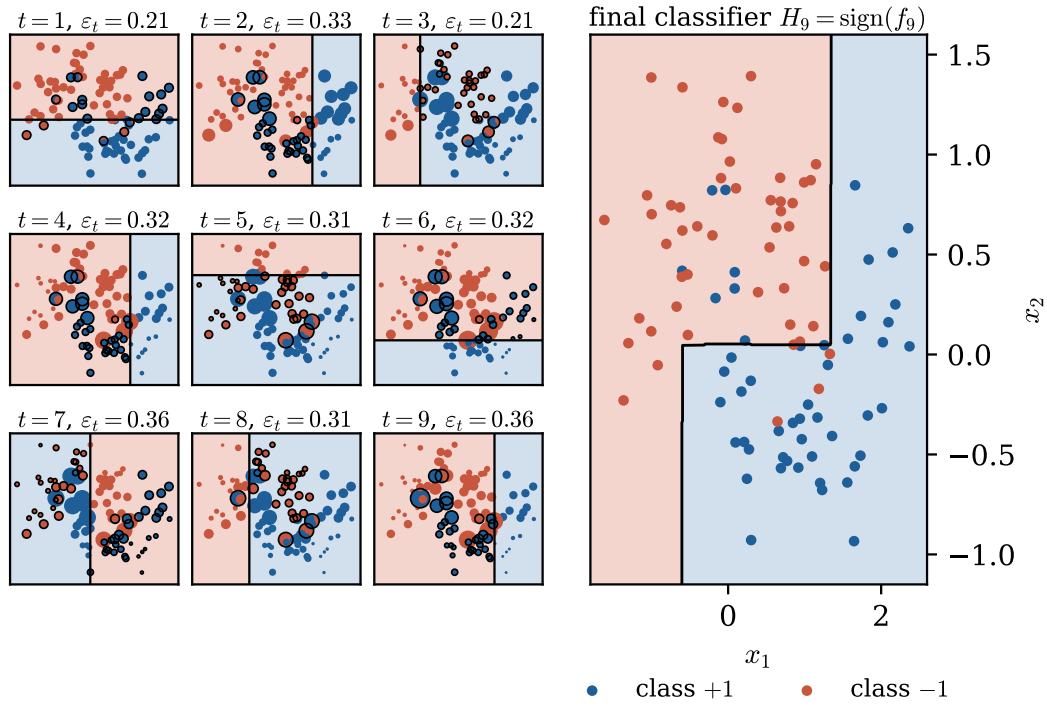


Figure 97: The first nine rounds of AdaBoost with decision stumps on a binary classification task ($m = 100$ points, class +1 in blue, class -1 in red). Left: the stump h_t learned at round t (the background color is its prediction) and its weighted error ε_t ; the area of each point is proportional to its weight $w_i^{(t)}$, and the points misclassified by h_t are circled in black: their weights increase at the next round. Right: the decision boundary of the final classifier $H_9 = \text{sign}(\sum_{t=1}^9 \alpha_t h_t)$, which is no longer a stump (training error 0.10). [code: make_figures_ch09.py: fig_adaboost_iterations]

Proposition 10.3: Empirical error bound of AdaBoost

The empirical error of the classifier returned by AdaBoost satisfies:

$$\mathcal{R}_S(H_T) \leq \exp \left[-2 \sum_{t=1}^T \left(\frac{1}{2} - \varepsilon_t \right)^2 \right].$$

Furthermore, if there exists $\gamma > 0$ such that, for all $t \in \llbracket 1, T \rrbracket$, $\gamma \leq \frac{1}{2} - \varepsilon_t$, then:

$$\mathcal{R}_S(H_T) \leq \exp(-2\gamma^2 T).$$

Proof. First, we recall that the exponential function is an upper bound of the 0-1 loss, i.e. for any real-valued function f ,

$$\forall (\mathbf{x}, y), \quad \mathbb{1}_{\{yf(\mathbf{x}) \leq 0\}} \leq \exp(-yf(\mathbf{x})).$$

We can then upper bound the empirical risk $\mathcal{R}_S(H_T)$, where $H_T = \text{sign}(f_T)$ with $f_T = \sum_{t=1}^T \alpha_t h_t$:

$$\begin{aligned}\mathcal{R}_S(H_T) &\leq \frac{1}{m} \sum_{i=1}^m \mathbb{1}_{\{y_i f_T(\mathbf{x}_i) \leq 0\}}, \\ &\leq \frac{1}{m} \sum_{i=1}^m \exp(-y_i f_T(\mathbf{x}_i)).\end{aligned}$$

We can now express this last sum using the normalization factors Z_t and the weights $w_i^{(T+1)}$. Indeed, we have:

$$\begin{aligned}w_i^{(t+1)} &= w_i^{(t)} \frac{\exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{Z_t}, \\ &\quad \downarrow \text{by induction, using } w_i^{(1)} = 1/m \\ &= \frac{1}{m} \times \frac{\exp\left(-y_i \sum_{s=1}^t \alpha_s h_s(\mathbf{x}_i)\right)}{\prod_{s=1}^t Z_s}.\end{aligned}$$

Thus, the empirical risk can be upper bounded as follows:

$$\begin{aligned}\mathcal{R}_S(H_T) &\leq \frac{1}{m} \sum_{i=1}^m \exp(-y_i f_T(\mathbf{x}_i)), \\ &\quad \downarrow \text{using the definition of } f_T \text{ and the previous relation with } t = T \\ &= \frac{1}{m} \sum_{i=1}^m \left(m \prod_{t=1}^T Z_t \right) w_i^{(T+1)}, \\ &\quad \downarrow \text{the weights } w_i^{(T+1)} \text{ sum to 1} \\ &= \prod_{t=1}^T Z_t.\end{aligned}$$

We now focus on the normalization factor and show how it can be written as a function of the weighted classification error ε_t for all $t \in \llbracket 1, T \rrbracket$:

$$\begin{aligned}Z_t &= \sum_{i=1}^m w_i^{(t)} \exp(-\alpha_t y_i h_t(\mathbf{x}_i)), \\ &= \sum_{i: y_i h_t(\mathbf{x}_i)=1} w_i^{(t)} \exp(-\alpha_t) + \sum_{i: y_i h_t(\mathbf{x}_i)=-1} w_i^{(t)} \exp(\alpha_t), \\ &= (1 - \varepsilon_t) \exp(-\alpha_t) + \varepsilon_t \exp(\alpha_t), \\ &\quad \downarrow \text{using the value of } \alpha_t \\ &= (1 - \varepsilon_t) \sqrt{\frac{\varepsilon_t}{1 - \varepsilon_t}} + \varepsilon_t \sqrt{\frac{1 - \varepsilon_t}{\varepsilon_t}}, \\ &= 2\sqrt{\varepsilon_t(1 - \varepsilon_t)}.\end{aligned}$$

As the empirical risk $\mathcal{R}_S(H_T)$ is upper bounded by the product of the normalization factors, we directly obtain:

$$\begin{aligned}
\mathcal{R}_S(H_T) &\leq \prod_{t=1}^T Z_t, \\
&= \prod_{t=1}^T 2\sqrt{\varepsilon_t(1-\varepsilon_t)}, \\
&\quad \downarrow \text{ using } 4\varepsilon_t(1-\varepsilon_t) = 1 - 4\left(\frac{1}{2} - \varepsilon_t\right)^2 \\
&= \prod_{t=1}^T \sqrt{1 - 4\left(\frac{1}{2} - \varepsilon_t\right)^2}, \\
&\quad \downarrow \text{ for all } x \in \mathbb{R}, 1 - x \leq \exp(-x) \\
&\leq \prod_{t=1}^T \exp\left[-2\left(\frac{1}{2} - \varepsilon_t\right)^2\right], \\
&\quad \downarrow \text{ property of the exponential} \\
&= \exp\left[-2\sum_{t=1}^T \left(\frac{1}{2} - \varepsilon_t\right)^2\right].
\end{aligned}$$

This ends the first part of the proof. Furthermore, if for all t , $\gamma \leq \frac{1}{2} - \varepsilon_t$, then:

$$\begin{aligned}
\mathcal{R}_S(H_T) &\leq \exp\left[-2\sum_{t=1}^T \left(\frac{1}{2} - \varepsilon_t\right)^2\right], \\
&\quad \downarrow \text{ using the assumption} \\
&\leq \exp\left[-2\sum_{t=1}^T \gamma^2\right], \\
&= \exp\left[-2T\gamma^2\right].
\end{aligned}$$

□

In other words, as soon as each weak learner is slightly better than random guessing, the training error of AdaBoost decreases exponentially fast with the number of rounds. Since the training error of H_T is a multiple of $1/m$, it even becomes exactly 0 as soon as the bound is smaller than $1/m$, *i.e.* after at most $\lfloor \ln(m)/(2\gamma^2) \rfloor + 1$ rounds. This behavior is illustrated in Figure 98.

Remark Proposition 10.3 only deals with the *training* error. Surprisingly, the test error of AdaBoost often keeps decreasing, or at least does not increase, long after the training error has reached 0 (in the left panel of Figure 98, the test error does not increase during the 300 rounds that follow): the complexity of f_T seems to grow with T , but the generalization does not degrade. The explanation proposed by Schapire *et al.* Schapire et al. (1998) is that AdaBoost keeps increasing the *margins* $y_i f_T(\mathbf{x}_i) / \sum_t \alpha_t$ of the training examples, which leads to generalization bounds independent of T . We come back to this question with PAC-Bayesian tools in Section 15.6.

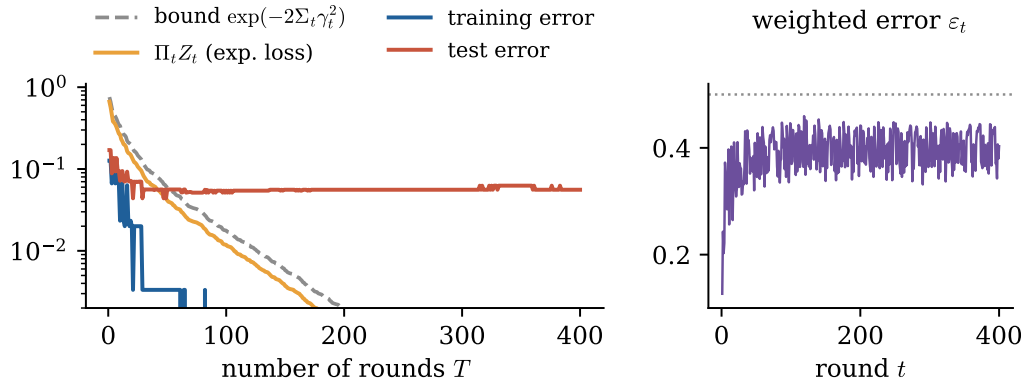


Figure 98: AdaBoost with decision stumps on the two-moons dataset ($m = 300$ training examples). Left (logarithmic scale): the training error is always below the empirical exponential loss $\prod_t Z_t = \frac{1}{m} \sum_i e^{-y_i f_T(\mathbf{x}_i)}$, itself below the bound of Proposition 10.3 with $\gamma_t = \frac{1}{2} - \varepsilon_t$; the training error reaches 0 in less than 100 rounds (and then disappears from the plot), whereas the test error stabilizes. Right: the weighted errors ε_t of the stumps increase, because the examples that are still misclassified are the hardest ones, but they remain below $1/2$. [code: `make_figures_ch09.py: fig_adaboost_bound`]

Pitfall: AdaBoost and label noise The weight of a misclassified example is multiplied by $e^{\alpha_t} > 1$ at each round. A mislabeled example, which the weak learners consistently “misclassify”, thus sees its weight grow exponentially, and AdaBoost ends up spending most of its effort on outliers. This is a direct consequence of the exponential loss, which penalizes large negative margins much more than the logistic loss does. On noisy data, prefer gradient boosting with the logistic loss, early stopping, or a small learning rate (Section 10.3).

We did not explain earlier where the expression of α_t comes from, but the answer lies in the previous proof. Indeed, α_t is chosen to minimize the upper bound $\prod_t Z_t$ of the empirical error. Thus, it is chosen to minimize the function:

$$\varphi : \alpha \mapsto (1 - \varepsilon_t) \exp(-\alpha) + \varepsilon_t \exp(\alpha).$$

This function is (strictly) convex as a positive combination of two (strictly) convex functions. So it reaches its minimum at a single value α_t , which satisfies the first-order optimality condition (Euler’s equation):

$$\begin{aligned} \varphi'(\alpha_t) &= 0, \\ -(1 - \varepsilon_t) \exp(-\alpha_t) + \varepsilon_t \exp(\alpha_t) &= 0, \\ (1 - \varepsilon_t) \exp(-\alpha_t) &= \varepsilon_t \exp(\alpha_t), \\ \frac{1 - \varepsilon_t}{\varepsilon_t} &= \exp(2\alpha_t), \\ \alpha_t &= \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right). \end{aligned}$$

Note also that the analysis extends to base hypotheses h_t taking real values in

the range $[-1, +1]$: in this case, ε_t is replaced by $\frac{1}{2} \left(1 - \sum_{i=1}^m w_i^{(t)} y_i h_t(\mathbf{x}_i)\right)$, and the expression of Z_t becomes an upper bound obtained by convexity of the exponential.

Link with gradient descent We can draw a link between AdaBoost and a gradient descent algorithm called *coordinate descent*: we just have to change our point of view on the algorithm.

Let us consider a (finite) family of base classifiers $h_1, h_2, \dots, h_N$ and a function f_{α} which depends on a vector $\alpha \in \mathbb{R}^N$ and is defined by

$$f_{\alpha} = \sum_{l=1}^N \alpha_l h_l.$$

Let us also consider a set of m labeled examples $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$. We aim to solve the following optimization problem:

$$\min_{\alpha \in \mathbb{R}^N} F(\alpha) = \min_{\alpha \in \mathbb{R}^N} \frac{1}{m} \sum_{i=1}^m \exp(-y_i f_{\alpha}(\mathbf{x}_i)) = \min_{\alpha \in \mathbb{R}^N} \frac{1}{m} \sum_{i=1}^m \exp\left(-y_i \sum_{l=1}^N \alpha_l h_l(\mathbf{x}_i)\right).$$

We see that the objective function F is exactly the upper bound of the empirical risk of $\text{sign}(f_{\alpha})$ used in the proof above. F is also a convex function of α , as a sum of convex functions.

The idea of coordinate descent is to update only one coordinate of the vector we are looking for at each iteration of the descent. More precisely, starting from a vector $\alpha^{(k-1)}$ at a given iteration k of the descent,

$$\alpha^{(k-1)} = \left(\alpha_1^{(k-1)}, \alpha_2^{(k-1)}, \dots, \alpha_l^{(k-1)}, \dots, \alpha_N^{(k-1)}\right),$$

we change only one coordinate of this vector, say the l -th one, to get the new vector $\alpha^{(k)}$, so that

$$\forall j \neq l, \quad \alpha_j^{(k)} = \alpha_j^{(k-1)} \quad \text{and} \quad \alpha_l^{(k)} = \alpha_l^{(k-1)} + \rho, \quad \text{i.e.} \quad \alpha^{(k)} = \alpha^{(k-1)} + \rho \mathbf{e}_l,$$

where ρ is a step size and $\mathbf{e}_l$ is the l -th vector of the canonical basis, *i.e.* the direction (or coordinate) that is updated.

Let us consider the function g_k defined at iteration k by $g_k = \sum_{l=1}^N \alpha_l^{(k)} h_l$. Then, the coordinate descent update coincides with the update $g_k = g_{k-1} + \rho h_l$. Thus, since both algorithms start with $g_0 = 0$, to show that AdaBoost coincides with coordinate descent applied to F , it suffices to show that, at every iteration k , coordinate descent selects the same base classifier and the same step ρ as AdaBoost, assuming that the weak learner of AdaBoost returns the classifier of the family with the smallest weighted error.

We assume by induction that this holds up to iteration $k-1$, which implies the equality $g_{k-1} = f_{k-1}$ (the function f built by AdaBoost after $k-1$ rounds), and we show that it also holds at iteration k .

At each iteration k , we select the steepest descent direction, that is, the direction $\mathbf{e}_l$ along which the directional derivative of the objective function F is the most negative, and we select the best step along that direction, that is, we choose ρ to minimize $F(\boldsymbol{\alpha}^{(k-1)} + \rho \mathbf{e}_l)$.

We first need to introduce quantities similar to those of the boosting algorithm. We denote by $\tilde{w}_i^{(k)}$ the weight of the i -th example at iteration k :

$$\tilde{w}_i^{(k)} = \frac{\exp\left(-y_i \sum_{l=1}^N \alpha_l^{(k-1)} h_l(\mathbf{x}_i)\right)}{\tilde{Z}_k} = \frac{\exp(-y_i g_{k-1}(\mathbf{x}_i))}{\tilde{Z}_k},$$

where $\tilde{Z}_k = \sum_{i=1}^m \exp(-y_i g_{k-1}(\mathbf{x}_i))$ is a normalization factor, such that the weights sum to 1. Since $g_{k-1} = f_{k-1}$, the relation obtained in the proof of Proposition 10.3 shows that $\mathbf{w}^{(k)} = \tilde{\mathbf{w}}^{(k)}$.

We also define the weighted error of a base classifier h_l with respect to the distribution $\tilde{\mathbf{w}}^{(k)}$ as:

$$\tilde{\varepsilon}_l^{(k)} = \mathbb{E}_{i \sim \tilde{\mathbf{w}}^{(k)}} \left[\mathbb{1}_{\{y_i h_l(\mathbf{x}_i) < 0\}} \right] = \sum_{i=1}^m \tilde{w}_i^{(k)} \mathbb{1}_{\{y_i h_l(\mathbf{x}_i) < 0\}}.$$

The directional derivative of F at $\boldsymbol{\alpha}^{(k-1)}$ along $\mathbf{e}_l$ is denoted by $F'(\boldsymbol{\alpha}^{(k-1)}, \mathbf{e}_l)$ and defined by

$$F'(\boldsymbol{\alpha}^{(k-1)}, \mathbf{e}_l) = \lim_{\rho \rightarrow 0} \frac{F(\boldsymbol{\alpha}^{(k-1)} + \rho \mathbf{e}_l) - F(\boldsymbol{\alpha}^{(k-1)})}{\rho}.$$

Since $F(\boldsymbol{\alpha}^{(k-1)} + \rho \mathbf{e}_l) = \frac{1}{m} \sum_{i=1}^m \exp\left(-y_i \sum_{j=1}^N \alpha_j^{(k-1)} h_j(\mathbf{x}_i) - \rho y_i h_l(\mathbf{x}_i)\right)$, the directional derivative can be expressed as:

$$\begin{aligned} F'(\boldsymbol{\alpha}^{(k-1)}, \mathbf{e}_l) &= -\frac{1}{m} \sum_{i=1}^m y_i h_l(\mathbf{x}_i) \exp\left(-y_i \sum_{j=1}^N \alpha_j^{(k-1)} h_j(\mathbf{x}_i)\right), \\ &\quad \downarrow \text{ we use the relation between } \tilde{Z}_k \text{ and the weights } \tilde{w}_i^{(k)} \\ &= -\frac{1}{m} \sum_{i=1}^m y_i h_l(\mathbf{x}_i) \tilde{w}_i^{(k)} \tilde{Z}_k, \\ &\quad \downarrow \text{ we split the sum according to the sign of } y_i h_l(\mathbf{x}_i) \\ &= -\frac{\tilde{Z}_k}{m} \left[\sum_{i: y_i h_l(\mathbf{x}_i) = 1} \tilde{w}_i^{(k)} - \sum_{i: y_i h_l(\mathbf{x}_i) = -1} \tilde{w}_i^{(k)} \right], \\ &\quad \downarrow \text{ using the definition of the error} \\ &= -\frac{\tilde{Z}_k}{m} \left[1 - \tilde{\varepsilon}_l^{(k)} - \tilde{\varepsilon}_l^{(k)} \right], \\ &= (2\tilde{\varepsilon}_l^{(k)} - 1) \frac{\tilde{Z}_k}{m}. \end{aligned}$$

The steepest descent direction is therefore the one that minimizes $\tilde{\varepsilon}_l^{(k)}$. So the base classifier h_l selected at iteration k is the one with the smallest weighted error on

the sample S with respect to the distribution $\tilde{\mathbf{w}}^{(k)} = \mathbf{w}^{(k)}$. This is exactly the choice made by AdaBoost at the k -th round.

It remains to see how the step size is computed along the chosen direction $\mathbf{e}_l$. The step size ρ is the solution of the following convex optimization problem:

$$\min_{\rho \in \mathbb{R}} F(\boldsymbol{\alpha}^{(k-1)} + \rho \mathbf{e}_l).$$

To minimize this quantity, it is enough to look for the value of ρ at which its derivative vanishes:

$$\begin{aligned} \frac{d}{d\rho} F(\boldsymbol{\alpha}^{(k-1)} + \rho \mathbf{e}_l) &= 0, \\ -\frac{1}{m} \sum_{i=1}^m y_i h_l(\mathbf{x}_i) \exp \left(-y_i \sum_{j=1}^N \alpha_j^{(k-1)} h_j(\mathbf{x}_i) - \rho y_i h_l(\mathbf{x}_i) \right) &= 0, \\ \sum_{i=1}^m y_i h_l(\mathbf{x}_i) \tilde{w}_i^{(k)} \tilde{Z}_k \exp(-\rho y_i h_l(\mathbf{x}_i)) &= 0, \\ \sum_{i=1}^m y_i h_l(\mathbf{x}_i) \tilde{w}_i^{(k)} \exp(-\rho y_i h_l(\mathbf{x}_i)) &= 0, \\ (1 - \tilde{\varepsilon}_l^{(k)}) \exp(-\rho) - \tilde{\varepsilon}_l^{(k)} \exp(\rho) &= 0, \\ \rho &= \frac{1}{2} \ln \left(\frac{1 - \tilde{\varepsilon}_l^{(k)}}{\tilde{\varepsilon}_l^{(k)}} \right). \end{aligned}$$

This proves that the step size chosen by coordinate descent coincides with the weight α_k assigned by AdaBoost to the classifier chosen at the k -th round. Thus, coordinate descent applied to the exponential objective F coincides exactly with AdaBoost, and F can be viewed as the objective function that AdaBoost seeks to minimize.

10.2.3 Stacking

Stacked generalization, or *stacking* Wolpert (1992); Džeroski and Ženko (2004), is another ensemble method, which works rather differently from bagging and boosting. Although the idea is still to combine models in order to improve performance, stacking learns a meta-model on top of several sub-models.

The aim of this ensemble method is to learn how different models, with different predictive capabilities, can be optimally combined. It differs from the two previous procedures in two ways: (i) the models can all be different (for instance, we can combine an SVM with a decision tree) and they are learned using the same dataset; (ii) the weights of the models are not learned iteratively. All the models are learned independently, and a meta-model is then used to learn the optimal way to combine the base learners.

Its architecture involves at least two different learners, and the procedure consists of two steps:

1. learn a set of base learners on the training data;

2. use the predictions made by each base learner as new features for the meta-model, which then learns a good combination of these predictions.

As the reader may have noticed, while each sub-model h_t works with the initial features $\mathbf{x}$, the meta-model works with the predictions of the sub-models, *i.e.* the features of the meta-model are $(h_1(\mathbf{x}), h_2(\mathbf{x}), \dots, h_T(\mathbf{x}))$. Note that we do not specify the output of h_t : it can be a real value, a binary output, a probability, *etc.*

At first glance, it seems rather simple to learn this ensemble model: all we have to do is train our sub-models on the training data and then reuse these data to learn our meta-model. However, this method is not the best in terms of generalization performance. Indeed, by using the same data to learn our sub-models and our meta-model, we increase the risk of over-fitting: the meta-model would learn to trust the sub-models that best memorize the training data. As a result, the data we give to the meta-model should not be the data used to learn the sub-models, but rather data used to validate them, *i.e.* a held-out validation set, resulting in a *training-validation* procedure.

We can go a little further and learn what is called a **Super Learner** [Van der Laan et al. \(2007\)](#), which can be seen as a generalization of stacking relying on a k -fold cross-validation procedure, as depicted in Figure 99.¹⁸

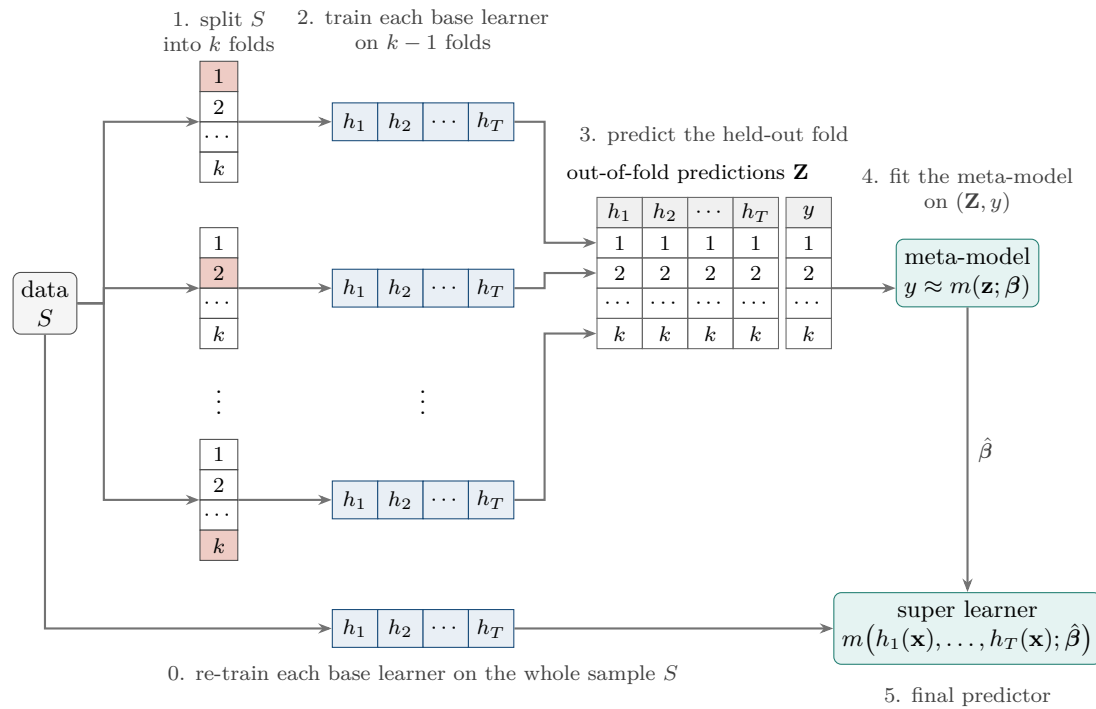


Figure 99: The Super Learner: stacking with a k -fold cross-validation procedure. Each base learner $h_1, \dots, h_T$ is trained k times, each time on $k - 1$ folds, and predicts the held-out fold (shaded). These out-of-fold predictions form a new design matrix $\mathbf{Z}$, on which the meta-model $m(\mathbf{z}; \beta)$ is fitted. The final predictor combines, with the learned parameters $\hat{\beta}$, the base learners re-trained on the whole sample.

The procedure can be described as follows:

1. Split the training data into k folds (as in a k -fold cross-validation procedure).

¹⁸This figure is adapted from slide 17 of the [Super Learner course](#) of the van der Laan lab.

2. Train each base learner on $k-1$ folds and compute its predictions on the remaining fold.
3. Repeat the previous step, changing the *validation* fold each time, until every fold has been used as the *validation* fold. Every training example then has a prediction from each base learner, made by a model that has not seen it: these *out-of-fold* predictions form the new representation of the data.
4. Use this new representation of the data to learn the parameters of the meta-model (the base learners are then usually re-trained on the whole training set).
5. Evaluate the performance of the whole model on the test data.

A legitimate question concerns the classic cross-validation procedure. How can we use our data not only to learn our model, but also to ensure that we have the best base learners, and therefore a good ensemble model? How can we tune the hyper-parameters of our base learners? Is this really necessary?

We leave these questions open, so that the reader can draw on the previous explanations to discuss possible answers (see also Section 11). We end our presentation of ensemble methods with the one that is certainly the best known and most widely used in industrial practice: *gradient boosting*.

10.3 Gradient Boosting

The AdaBoost algorithm (see Section 10.2.2) is based on the exponential loss. However, this loss is not suited to all settings, and it is sometimes better to use other losses, depending on the model we want to learn or on the application. This motivation leads us to a more general boosting algorithm, *gradient boosting* Friedman (2001).

Generalities Unlike the AdaBoost algorithm Freund and Schapire (1999), as usually presented, gradient boosting performs an optimization in the *function* space rather than in the *parameter* space. At each iteration, a weak learner h_t is learned using the *residuals* (or errors) of the linear combination of the previous models. The linear combination H_t at iteration t is defined as follows:

$$H_t = H_{t-1} + \alpha_t h_t, \quad (10.1)$$

where H_{t-1} is the linear combination of the first $t-1$ models and α_t is the weight given to the t -th weak learner. The weak learners are trained on the *pseudo-residuals* r_i of the current model. These pseudo-residuals are given by the negative gradient $-g_t$ of the loss function ℓ with respect to the current prediction $H_{t-1}(\mathbf{x}_i)$:

$$r_i = -g_t(\mathbf{x}_i) = - \left[\frac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)} \right].$$

Once the pseudo-residuals r_i are computed, the following optimization problem is solved:

$$(h_t, \alpha_t) = \arg \min_{\alpha, h} \sum_{i=1}^m (r_i - \alpha h(\mathbf{x}_i))^2.$$

Finally, the update rule (10.1) is applied. In practice, the weight α_t is often refined by a line search on the loss ℓ itself, as in Algorithm 9, and multiplied by a small *learning rate* (or *shrinkage*) $\nu \in (0, 1]$ to reduce over-fitting.

In a nutshell Think of a golfer who does not reach the hole in one shot: each new stroke starts from where the ball stopped, and aims at the remaining distance. With the squared loss $\frac{1}{2}(y - H)^2$, the pseudo-residuals are exactly the residuals $y_i - H_{t-1}(\mathbf{x}_i)$: each new weak learner is trained to predict *what the current model still gets wrong*, and is added to it. Figure 100 shows the first three rounds on a one-dimensional regression problem, with decision stumps as weak learners. For other losses, the pseudo-residuals are “generalized residuals”: they indicate in which direction, and how strongly, the prediction at each training point should move to decrease the loss.

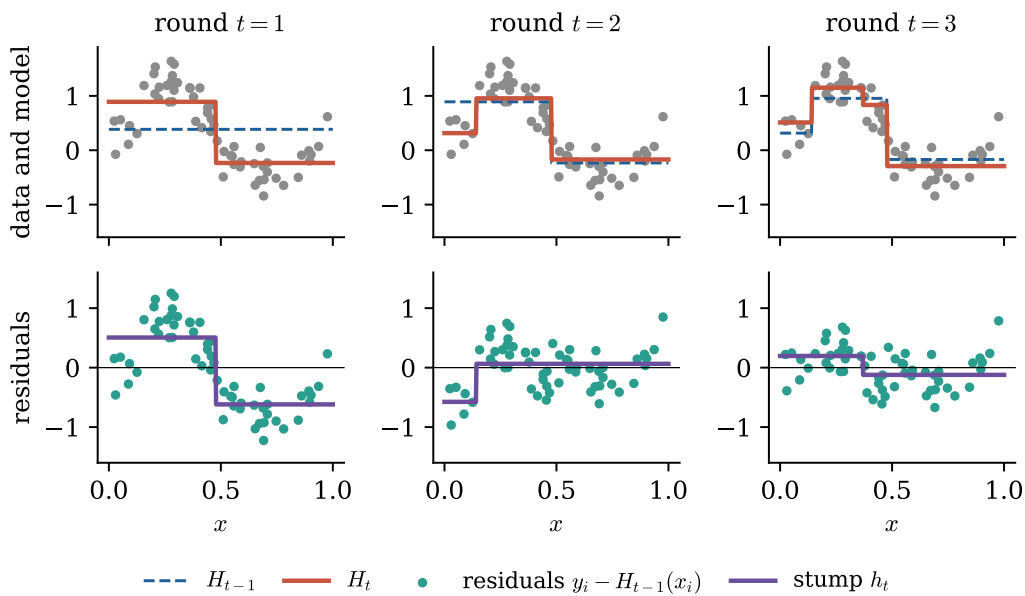


Figure 100: Gradient boosting with the squared loss and decision stumps on a 1D regression problem (H_0 is the mean of the y_i , $\nu = 1$). Top: the model before (H_{t-1} , dashed) and after (H_t , solid) round t . Bottom: the residuals of H_{t-1} and the stump h_t fitted to them by least squares. For the squared loss, the line search gives $\alpha_t = 1$, so that $H_t = H_{t-1} + h_t$; the residuals shrink from one round to the next and the model gets progressively closer to the data. [code: `make_figures_ch09.py: fig_gb_residuals`]

This algorithm was first developed for classification and regression trees, and most of the work and libraries, such as **XGBoost** Chen and Guestrin (2016), use decision trees as weak learners. To be weak learners, these trees are mainly decision stumps or trees with a small depth. Unlike AdaBoost, gradient boosting makes it possible to use custom losses. The procedure is summarized in Algorithm 9, where the weak learners $h_{\mathbf{a}}$ are parameterized by a vector $\mathbf{a}$.

Example 10.2. Let us compute the pseudo-residuals for two different losses ℓ : the *squared loss* for a regression task, and the *logistic loss* for a classification task.

- Using the squared loss $\ell(y, H) = (y - H)^2$, the pseudo-residuals are defined by:

Algorithm 9: Gradient Boosting **Friedman (2001)**

Inputs : Training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, number of models T , a loss ℓ

Output: a model $H_T = H_0 + \sum_{t=1}^T \alpha_t h_{\mathbf{a}_t}$

- 1 Initial hypothesis: $H_0(\mathbf{x}) = \arg \min_{\rho \in \mathbb{R}} \sum_{i=1}^m \ell(y_i, \rho)$ for all $\mathbf{x}$;
 - 2 **for** $t = 1$ **to** T **do**
 - 3 compute the pseudo-residuals: $\tilde{y}_i = -\frac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)}, \forall i = 1, \dots, m$;
 - 4 fit the pseudo-residuals to learn the new hypothesis:
 $\mathbf{a}_t = \arg \min_{\mathbf{a}} \sum_{i=1}^m (\tilde{y}_i - h_{\mathbf{a}}(\mathbf{x}_i))^2$;
 - 5 learn the weight of the hypothesis $h_{\mathbf{a}_t}$:
 $\alpha_t = \arg \min_{\alpha \in \mathbb{R}^+} \sum_{i=1}^m \ell(y_i, H_{t-1}(\mathbf{x}_i) + \alpha h_{\mathbf{a}_t}(\mathbf{x}_i))$;
 - 6 update: $H_t = H_{t-1} + \alpha_t h_{\mathbf{a}_t}$;
-

$$\begin{aligned}\tilde{y} &= -\frac{\partial \ell(y, H_{t-1}(\mathbf{x}))}{\partial H_{t-1}(\mathbf{x})}, \\ &= -\frac{\partial (y - H_{t-1}(\mathbf{x}))^2}{\partial H_{t-1}(\mathbf{x})}, \\ &= 2(y - H_{t-1}(\mathbf{x})).\end{aligned}$$

The pseudo-residual is just twice the difference between the true value and the predicted value, *i.e.* (up to the factor 2) the usual residual (remember that this loss is mainly used in the regression setting).

- Using the logistic loss $\ell(y, H) = \ln(1 + \exp(-2yH))$ for classification with labels $y \in \{-1, +1\}$ (this is the convention used in **Friedman (2001)**), the pseudo-residuals are defined by:

$$\begin{aligned}\tilde{y} &= -\frac{\partial \ell(y, H_{t-1}(\mathbf{x}))}{\partial H_{t-1}(\mathbf{x})}, \\ &= -\frac{\partial \ln(1 + \exp(-2yH_{t-1}(\mathbf{x})))}{\partial H_{t-1}(\mathbf{x})}, \\ &= \frac{2y \exp(-2yH_{t-1}(\mathbf{x}))}{1 + \exp(-2yH_{t-1}(\mathbf{x}))}, \\ &= \frac{2y}{1 + \exp(2yH_{t-1}(\mathbf{x}))}.\end{aligned}$$

Note that the pseudo-residual vanishes when $yH_{t-1}(\mathbf{x})$ is large, *i.e.* when the example is already correctly classified with a large margin.

This is not the most commonly used version of this loss. Since $H_{t-1}(\mathbf{x})$ is a real value, we rather pass this output through the logistic function in order to obtain the probability of belonging to the positive class, and use labels $y \in \{0, 1\}$. In such a case, the pseudo-residuals are given by:

Algorithm 10: Gradient boosting seen as a steepest gradient descent in function space [Friedman \(2001\)](#)

Inputs : Training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, number of models T , a loss ℓ
Output: a model $H_T = H_0 + \sum_{t=1}^T \alpha_t h_{\mathbf{a}_t}$
1 **Initialization:** $H_0(\mathbf{x}) = \arg \min_{\rho \in \mathbb{R}} \sum_{i=1}^m \ell(y_i, \rho)$ for all $\mathbf{x}$;
2 **for** $t = 1$ **to** T **do**
3 **Compute the negative gradient:** $\tilde{y}_i = -\frac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)}, \forall i = 1, \dots, m$;
4 **Learn a model that approximates this direction:**
 $\mathbf{a}_t = \arg \min_{\mathbf{a}} \sum_{i=1}^m (\tilde{y}_i - h_{\mathbf{a}}(\mathbf{x}_i))^2$;
5 **Find the optimal step** for $h_{\mathbf{a}_t}$: $\alpha_t = \arg \min_{\alpha \in \mathbb{R}^+} \sum_{i=1}^m \ell(y_i, H_{t-1}(\mathbf{x}_i) + \alpha h_{\mathbf{a}_t}(\mathbf{x}_i))$;
6 **Update:** $H_t = H_{t-1} + \alpha_t h_{\mathbf{a}_t}$;

$$\begin{aligned} \tilde{y} &= -\frac{\partial \ell(y, H_{t-1}(\mathbf{x}))}{\partial H_{t-1}(\mathbf{x})}, \\ &= -\frac{\partial [-y \ln(\sigma(H_{t-1}(\mathbf{x}))) - (1-y) \ln(1 - \sigma(H_{t-1}(\mathbf{x})))]}{\partial H_{t-1}(\mathbf{x})}, \\ &\quad \downarrow \text{same computation as for the logistic regression, using } \sigma' = \sigma(1 - \sigma) \\ &= y - \sigma(H_{t-1}(\mathbf{x})), \end{aligned}$$

where σ denotes the logistic function $\sigma(x) = (1 + e^{-x})^{-1}$. The pseudo-residual is then the difference between the label and the predicted probability.

We can also draw a parallel between the gradient boosting algorithm and a gradient descent algorithm. This is quite natural, since the algorithm involves the gradient of the loss. Here, the parallel can be drawn with the steepest gradient descent with an optimal step size (computed by a line search), as shown in Algorithm 10: gradient boosting is a gradient descent in the space of functions, in which the gradient, only known at the training points, is approximated by a weak learner.

Note that gradient boosting, in its original presentation, uses the mean squared error in the step that fits the pseudo-residuals. However, we can choose any loss function we want to optimize, such as the exponential loss for classification tasks. Furthermore, the reader has to be careful: the loss ℓ we aim to optimize is the one used both to compute the pseudo-residuals and to find the weights of the base learners, whereas the squared loss is only used to fit the pseudo-residuals.

Regularization of gradient boosting Unlike bagging, boosting *can* over-fit when the number of weak learners T grows: each round decreases the training loss, and nothing prevents the model from eventually fitting the noise. The two main levers to control this are the number of rounds T , chosen by early stopping on a validation set, and the learning rate ν , which replaces the update by $H_t = H_{t-1} + \nu \alpha_t h_t$. Figure 101 shows their interplay: a smaller ν requires more rounds, but leads to a better minimum of the test error.

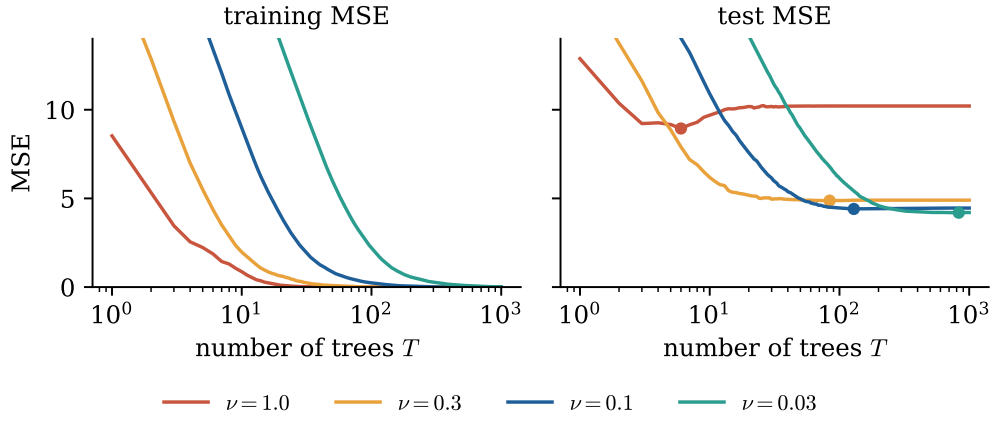


Figure 101: Effect of the learning rate ν in gradient boosting (squared loss, trees of depth 3, `make_friedman1` regression problem with 200 training examples). Left: the training error always goes to 0, all the faster as ν is large. Right: the test error reaches a minimum (dots) and then slightly increases or stagnates; with $\nu = 1$ the model quickly over-fits, whereas smaller values of ν reach a lower test error, at the price of more trees. [code: `make_figures_ch09.py`: `fig_gb_shrinkage`]

Remark : gradient boosting in practice Small steps generalize better: a value $\nu \in [0.01, 0.1]$ combined with early stopping is a robust default [Friedman \(2001\)](#), and ν and T should be tuned *together*. Friedman also showed that fitting each weak learner on a random subsample of the data (typically half of it), as in bagging, both speeds up the algorithm and improves its accuracy: this is *stochastic gradient boosting* [Friedman \(2002\)](#). Modern libraries such as XGBoost [Chen and Guestrin \(2016\)](#), presented below, LightGBM [Ke et al. \(2017\)](#) and CatBoost [Prokhorenkova et al. \(2018\)](#) add many algorithmic improvements (histograms of the features, sparsity handling, efficient treatment of categorical features, *etc.*). On tabular data, these tree ensembles remain, to this day, very hard to beat, even by deep neural networks [Grinsztajn et al. \(2022\)](#).

Presentation of XGBoost We finish this presentation of boosting by going back to decision trees and presenting XGBoost [Chen and Guestrin \(2016\)](#), a powerful and very fast tree-based boosting algorithm. We first explain how the weights of the leaves are computed, and then we explain the splitting criterion that is used. The main strength of this algorithm is that it can be used to optimize a risk based on any loss function, provided that this loss function is twice differentiable.

From now on, we consider a twice differentiable loss function ℓ and the following optimization problem:

$$\min_{f_1, \dots, f_T} \sum_{i=1}^m \ell(y_i, \hat{y}_i) + \Omega(f_1, \dots, f_T), \quad (10.2)$$

where $\hat{y}_i = \sum_{t=1}^T f_t(\mathbf{x}_i)$, T is the number of models and the f_t are the base learners (trees). The term $\Omega(f_1, \dots, f_T) = \sum_{t=1}^T \left(\beta \mathcal{L}_t + \frac{\lambda}{2} \sum_{j=1}^J (f_t^{(j)})^2 \right)$ is composed

of two regularization terms, used to control the number of leaves $\mathcal{L}_t$ of each tree and the weight $f_t^{(j)}$ of each leaf j of the t -th tree.

Recall that the models are learned in an additive manner, so let us denote by $\hat{y}_i^{(t-1)}$ the value predicted by the first $t-1$ trees, *i.e.* $\hat{y}_i^{(t-1)} = \sum_{s=1}^{t-1} f_s(\mathbf{x}_i) = H_{t-1}(\mathbf{x}_i)$.

Let us now have a look at the tree f_t learned at iteration t of our algorithm. For this purpose, we rewrite the objective (10.2) and aim to minimize, with respect to f_t , the quantity R_t defined as:

$$R_t = \sum_{i=1}^m \ell(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)) + \beta \mathcal{L}_t + \frac{\lambda}{2} \sum_{j=1}^{\mathcal{L}_t} (f_t^{(j)})^2. \quad (10.3)$$

In practice, the authors of [Chen and Guestrin \(2016\)](#) only consider a second-order approximation of the function they aim to optimize. This second-order approximation is taken with respect to the value predicted at the previous iteration, *i.e.* $\hat{y}_i^{(t-1)}$. Recall that, for any function u which is twice differentiable on an open set, we have, for any x and any small ε :

$$u(x + \varepsilon) \simeq u(x) + u'(x)\varepsilon + \frac{1}{2}u''(x)\varepsilon^2.$$

We respectively denote by $g(\mathbf{x}_i)$ and $h(\mathbf{x}_i)$ the first- and second-order derivatives of the function $\ell(y_i, \cdot)$ taken at $\hat{y}_i^{(t-1)}$. We can then approximate (10.3) as follows:

$$R_t \simeq \sum_{i=1}^m \left[\ell(y_i, \hat{y}_i^{(t-1)}) + f_t(\mathbf{x}_i)g(\mathbf{x}_i) + \frac{1}{2}f_t^2(\mathbf{x}_i)h(\mathbf{x}_i) \right] + \beta \mathcal{L}_t + \frac{\lambda}{2} \sum_{j=1}^{\mathcal{L}_t} (f_t^{(j)})^2. \quad (10.4)$$

Remember that we aim to learn the function f_t , *i.e.* the values $(f_t^{(j)})_{j=1, \dots, \mathcal{L}_t}$ of its leaves once the structure of the tree is fixed. Let us consider a leaf j and denote by I_j the set of indices i such that $\mathbf{x}_i$ falls into the leaf j , so that $f_t(\mathbf{x}_i) = f_t^{(j)}$ for all $i \in I_j$. Since the first term of (10.4) does not depend on f_t , the value $f_t^{(j)}$ shall minimize the following quantity V_j for a given index j :

$$V_j = \left(\sum_{i \in I_j} g(\mathbf{x}_i) \right) f_t^{(j)} + \frac{1}{2} \left(\lambda + \sum_{i \in I_j} h(\mathbf{x}_i) \right) (f_t^{(j)})^2. \quad (10.5)$$

This function is convex when $\lambda + \sum_{i \in I_j} h(\mathbf{x}_i) > 0$ (since $\lambda \geq 0$, this is the case as soon as the loss function ℓ is convex, so that $h(\mathbf{x}_i) \geq 0$, and $\lambda > 0$), and its minimum is then given by the solution of *Euler's equation*, *i.e.* the value $f_t^{(j)}$ at which the derivative vanishes. This solution is given by:

$$f_t^{(j)} = - \frac{\sum_{i \in I_j} g(\mathbf{x}_i)}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda}. \quad (10.6)$$

Example 10.3. Let us come back to our previous example, where we considered the *squared loss* and the *logistic loss*, and let us compute the weights of the leaves for these two losses.

For the squared loss $\ell(y, \hat{y}^{(t-1)}) = \frac{1}{2}(y - \hat{y}^{(t-1)})^2$, the gradient g with respect to the prediction is

$$g = \frac{\partial \ell}{\partial \hat{y}^{(t-1)}}(y, \hat{y}^{(t-1)}) = \hat{y}^{(t-1)} - y,$$

and the second-order derivative h with respect to the prediction is

$$h = \frac{\partial^2 \ell}{\partial (\hat{y}^{(t-1)})^2}(y, \hat{y}^{(t-1)}) = 1.$$

So the optimal value of a leaf $f_t^{(j)}$ is:

$$f_t^{(j)} = -\frac{\sum_{i \in I_j} g(\mathbf{x}_i)}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda} = \frac{\sum_{i \in I_j} (y_i - \hat{y}_i^{(t-1)})}{\lambda + |I_j|}.$$

Thus, when learning the first tree, with $\hat{y}_i^{(0)} = 0$ for all i and $\lambda = 0$, we find that the optimal score of a leaf is equal to the average of the target values of the instances in the leaf. For subsequent iterations, the optimal score of each leaf becomes the average of the residuals (shrunk towards 0 when $\lambda > 0$).

For the logistic loss $\ell(y, \hat{y}^{(t-1)}) = -\left(y \ln(p(\hat{y}^{(t-1)})) + (1 - y) \ln(1 - p(\hat{y}^{(t-1)}))\right)$, where $y \in \{0, 1\}$ and p is the logistic function, the gradient g with respect to the prediction is

$$g = \frac{\partial \ell}{\partial \hat{y}^{(t-1)}}(y, \hat{y}^{(t-1)}) = p(\hat{y}^{(t-1)}) - y,$$

and the second-order derivative h with respect to the prediction is

$$h = \frac{\partial^2 \ell}{\partial (\hat{y}^{(t-1)})^2}(y, \hat{y}^{(t-1)}) = p(\hat{y}^{(t-1)})(1 - p(\hat{y}^{(t-1)})).$$

So the optimal value of a leaf $f_t^{(j)}$ is:

$$f_t^{(j)} = -\frac{\sum_{i \in I_j} g(\mathbf{x}_i)}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda} = \frac{\sum_{i \in I_j} (y_i - p(\hat{y}_i^{(t-1)}))}{\lambda + \sum_{i \in I_j} p(\hat{y}_i^{(t-1)})(1 - p(\hat{y}_i^{(t-1)}))}.$$

Splitting criterion Let us now focus on the splitting criterion. Once the optimal weight (10.6) of each leaf is found, we can compute the optimal value V_j^* of the objective by plugging it into (10.5). We get:

$$\begin{aligned}
V_j^* &= - \underbrace{\left(\sum_{i \in I_j} g(\mathbf{x}_i) \right) \frac{\sum_{i \in I_j} g(\mathbf{x}_i)}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda}}_{\text{blue}} + \underbrace{\frac{1}{2} \left(\lambda + \sum_{i \in I_j} h(\mathbf{x}_i) \right) \left(- \frac{\sum_{i \in I_j} g(\mathbf{x}_i)}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda} \right)^2}_{\text{red}}, \\
&= - \frac{\left(\sum_{i \in I_j} g(\mathbf{x}_i) \right)^2}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda} + \frac{1}{2} \frac{\left(\sum_{i \in I_j} g(\mathbf{x}_i) \right)^2}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda}, \\
&= - \frac{1}{2} \frac{\left(\sum_{i \in I_j} g(\mathbf{x}_i) \right)^2}{\sum_{i \in I_j} h(\mathbf{x}_i) + \lambda}.
\end{aligned}$$

This formula is used to measure the quality of a leaf (the lower, the better). It can be seen as a generalization of impurity measures, such as the Gini impurity, to any loss function. Using this measure, the authors define their splitting criterion, *i.e.* the gain associated with the split of a leaf into a left and a right child, as follows:

$$\text{Gain} = \frac{1}{2} \left[\frac{\left(\sum_{i \in I_L} g(\mathbf{x}_i) \right)^2}{\sum_{i \in I_L} h(\mathbf{x}_i) + \lambda} + \frac{\left(\sum_{i \in I_R} g(\mathbf{x}_i) \right)^2}{\sum_{i \in I_R} h(\mathbf{x}_i) + \lambda} - \frac{\left(\sum_{i \in I} g(\mathbf{x}_i) \right)^2}{\sum_{i \in I} h(\mathbf{x}_i) + \lambda} \right] - \beta,$$

where $I = I_L \cup I_R$ is the set of examples of the leaf before the split, and the parameter β penalizes the creation of an additional leaf. The split with the largest gain is selected, and a leaf is not split if no split has a positive gain.

10.4 Metric Learning

All the algorithms presented so far rely, explicitly or implicitly, on a way to compare examples. The k -nearest neighbors algorithm (Section 8) predicts the label of a new point from the labels of its closest training examples, the k -means algorithm groups points that are close to each other, and kernel methods such as SVMs rely on a similarity function between examples. In all these cases, the performance of the algorithm crucially depends on the *representation* of the data and on the *distance* used to compare them. Most of the time, this distance is the Euclidean one, chosen by default, even though nothing guarantees that it is adapted to the task at hand.

Learning a good representation of the data is one of the key ingredients of modern machine learning: this is precisely what the hidden layers of neural networks do (Section 9). *Metric learning* addresses this question from a different angle: instead of learning a model on a fixed representation, it learns, from the data, the distance (or the similarity) used to compare the examples, in such a way that examples that should be considered as similar (*e.g.* examples of the same class) are close, while examples that should be considered as dissimilar are far apart. This section is largely inspired by the survey of Bellet, Habrard and Sebban [Bellet et al. \(2013\)](#), see also their book [Bellet et al. \(2015\)](#); we restrict ourselves to the most classical family of learned metrics, the Mahalanobis distances, and we will not go further than the LMNN algorithm.

10.4.1 Why Learn a Metric?

Let us recall what a distance is.

Definition 10.4: Distance and pseudo-distance

A *distance* (or *metric*) on a set $\mathcal{X}$ is a function $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$ such that, for all $\mathbf{x}, \mathbf{x}', \mathbf{x}'' \in \mathcal{X}$:

1. $d(\mathbf{x}, \mathbf{x}') = 0 \iff \mathbf{x} = \mathbf{x}'$ (separation)
2. $d(\mathbf{x}, \mathbf{x}') = d(\mathbf{x}', \mathbf{x})$ (symmetry)
3. $d(\mathbf{x}, \mathbf{x}'') \leq d(\mathbf{x}, \mathbf{x}') + d(\mathbf{x}', \mathbf{x}'')$ (triangle inequality)

A function satisfying these properties, except that the separation property is weakened into $d(\mathbf{x}, \mathbf{x}) = 0$ (two distinct points may be at distance 0), is called a *pseudo-distance*.

The Euclidean distance $d_2(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2$ gives the same importance to all the features and ignores their correlations. This causes at least three kinds of problems, which we have already met when we studied the k -NN algorithm:

- **scale:** a feature taking large values (*e.g.* an income in euros) dominates a feature taking small values (*e.g.* a height in meters), whatever their relevance for the task;
- **relevance:** non-informative, noisy features contribute to the distance as much as the informative ones, so that the nearest neighbors of a point may be determined by noise;
- **correlations:** two strongly correlated features count twice the same information.

Standardizing the features (Section 11) solves the first problem, but not the other two, since it does not use the labels. The idea of metric learning is to use the supervision to reshape the space, as illustrated in Figure 103 below: after learning, the neighbors of each point should mostly belong to its class, which directly improves the k -NN classifier.

10.4.2 Mahalanobis Distances

The most studied family of learned metrics is the family of (generalized) Mahalanobis distances. In what follows, $\mathbb{S}_+^d$ denotes the cone of symmetric positive semi-definite (PSD) matrices of size $d \times d$, and we write $\mathbf{M} \succeq 0$ for $\mathbf{M} \in \mathbb{S}_+^d$.

Definition 10.5: Mahalanobis distance

Let $\mathbf{M} \in \mathbb{S}_+^d$. The Mahalanobis distance associated with $\mathbf{M}$ is defined, for all $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$, by

$$d_{\mathbf{M}}(\mathbf{x}, \mathbf{x}') = \sqrt{(\mathbf{x} - \mathbf{x}')^\top \mathbf{M} (\mathbf{x} - \mathbf{x}')}.$$

A few remarks are in order:

- when $\mathbf{M} = \mathbf{I}$, we recover the Euclidean distance;

- when $\mathbf{M}$ is diagonal, $d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^d M_{jj}(x_j - x'_j)^2$ simply re-weights the features; standardizing the data amounts to choosing $M_{jj} = 1/s_j^2$, where s_j is the standard deviation of the j -th feature;
- when $\mathbf{M} = \Sigma^{-1}$ is the inverse of the covariance matrix of the data, we obtain the distance originally introduced by P. C. Mahalanobis, which takes the correlations between the features into account;
- a general matrix $\mathbf{M}$ both re-weights the features and takes their correlations into account.

The PSD assumption is essential: it ensures that $(\mathbf{x} - \mathbf{x}')^\top \mathbf{M}(\mathbf{x} - \mathbf{x}') \geq 0$, so that the square root is well defined, and it makes $d_{\mathbf{M}}$ a pseudo-distance. When $\mathbf{M}$ is positive definite, $d_{\mathbf{M}}$ is a distance; when $\mathbf{M}$ is only PSD, two distinct points $\mathbf{x}$ and $\mathbf{x}'$ such that $\mathbf{x} - \mathbf{x}'$ belongs to the kernel of $\mathbf{M}$ are at distance 0, which can be useful to discard irrelevant directions of the space. The following proposition gives a very convenient interpretation of Mahalanobis distances.

Proposition 10.4: Mahalanobis distances are Euclidean distances after a linear map

Let $\mathbf{M}$ be a symmetric matrix of size $d \times d$. Then $\mathbf{M} \succeq 0$ if and only if there exists a matrix $\mathbf{L} \in \mathbb{R}^{r \times d}$, with $r \geq \text{rank}(\mathbf{M})$, such that $\mathbf{M} = \mathbf{L}^\top \mathbf{L}$. In this case, for all $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$,

$$d_{\mathbf{M}}(\mathbf{x}, \mathbf{x}') = \|\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}'\|_2.$$

Proof. If $\mathbf{M} = \mathbf{L}^\top \mathbf{L}$, then $\mathbf{M}$ is symmetric and, for any $\mathbf{z} \in \mathbb{R}^d$, $\mathbf{z}^\top \mathbf{M} \mathbf{z} = \mathbf{z}^\top \mathbf{L}^\top \mathbf{L} \mathbf{z} = \|\mathbf{L}\mathbf{z}\|_2^2 \geq 0$, so $\mathbf{M} \succeq 0$. Conversely, if $\mathbf{M} \succeq 0$, the spectral theorem gives $\mathbf{M} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^\top$, where $\mathbf{U}$ is orthogonal and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_d)$ with $\lambda_k \geq 0$. Setting $\mathbf{L} = \mathbf{\Lambda}^{1/2} \mathbf{U}^\top$ gives $\mathbf{L}^\top \mathbf{L} = \mathbf{U} \mathbf{\Lambda}^{1/2} \mathbf{\Lambda}^{1/2} \mathbf{U}^\top = \mathbf{M}$; removing the rows of $\mathbf{L}$ associated with zero eigenvalues gives a matrix with $r = \text{rank}(\mathbf{M})$ rows. Finally, with $\mathbf{z} = \mathbf{x} - \mathbf{x}'$, we have $d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = \mathbf{z}^\top \mathbf{L}^\top \mathbf{L} \mathbf{z} = \|\mathbf{L}\mathbf{z}\|_2^2$. $\square$

In other words, **learning a Mahalanobis distance is equivalent to learning a linear transformation $\mathbf{x} \mapsto \mathbf{L}\mathbf{x}$** of the data and using the Euclidean distance in the new space. If $\mathbf{M}$ has a low rank $r < d$, this transformation also performs a (supervised) *dimensionality reduction*. Geometrically, the unit ball $\{\mathbf{x} : d_{\mathbf{M}}(\mathbf{x}, \mathbf{0}) \leq 1\}$ is an ellipsoid whose axes are given by the eigenvectors $\mathbf{u}_k$ of $\mathbf{M}$, with semi-axis lengths $1/\sqrt{\lambda_k}$ (see Figure 102).

Example 10.4. Let us consider $\mathbf{M} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$, whose eigenvalues are $\lambda_1 = 3$, associated with $\mathbf{u}_1 = \frac{1}{\sqrt{2}}(1, 1)^\top$, and $\lambda_2 = 1$, associated with $\mathbf{u}_2 = \frac{1}{\sqrt{2}}(1, -1)^\top$. Hence $\mathbf{M} \succeq 0$ (it is even positive definite).

- For $\mathbf{x} = (1, 0)^\top$ and $\mathbf{x}' = (0, 1)^\top$, $\mathbf{x} - \mathbf{x}' = (1, -1)^\top$ and $d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = 2 - 1 - 1 + 2 = 2$: the distance is equal to the Euclidean one, $\sqrt{2}$, because $\mathbf{x} - \mathbf{x}'$ is along $\mathbf{u}_2$.
- For $\mathbf{x} = (1, 1)^\top$ and $\mathbf{x}' = (0, 0)^\top$, $d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = 2 + 1 + 1 + 2 = 6$, so $d_{\mathbf{M}}(\mathbf{x}, \mathbf{x}') = \sqrt{6} = \sqrt{3} \times \sqrt{2}$: distances along $\mathbf{u}_1$ are stretched by a factor $\sqrt{\lambda_1} = \sqrt{3}$.

A possible choice of linear map is $\mathbf{L} = \mathbf{\Lambda}^{1/2} \mathbf{U}^\top = \frac{1}{\sqrt{2}} \begin{pmatrix} \sqrt{3} & \sqrt{3} \\ 1 & -1 \end{pmatrix}$, and one can check that $\mathbf{L}^\top \mathbf{L} = \mathbf{M}$.

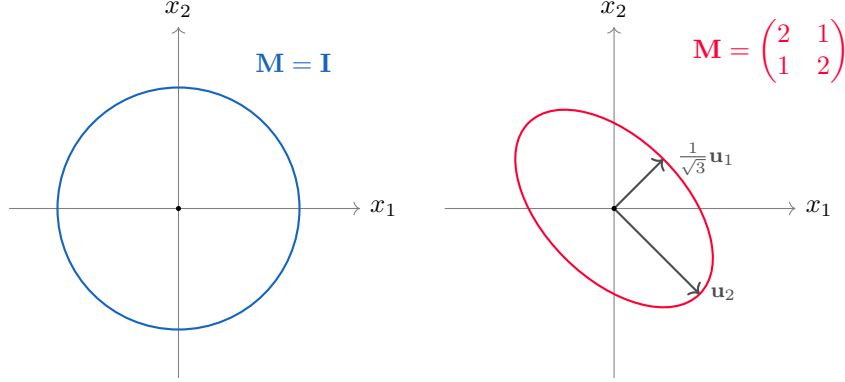


Figure 102: Unit balls $\{\mathbf{x} : d_{\mathbf{M}}(\mathbf{x}, \mathbf{0}) \leq 1\}$ of the Euclidean distance (left) and of the Mahalanobis distance of the example above (right). The ellipse is oriented along the eigenvectors of $\mathbf{M}$, with semi-axes $1/\sqrt{\lambda_k}$: the directions associated with large eigenvalues are the ones along which distances are stretched.

Why does it help k -NN and k -means? Since $d_{\mathbf{M}}(\mathbf{x}, \mathbf{x}') = \|\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}'\|_2$, running the k -NN algorithm with $d_{\mathbf{M}}$ is the same as running the usual (Euclidean) k -NN algorithm on the transformed data $\mathbf{L}\mathbf{x}_1, \dots, \mathbf{L}\mathbf{x}_m$. If $\mathbf{L}$ stretches the directions that discriminate the classes and shrinks the irrelevant ones, the neighborhood of a point contains more examples of its own class, and the majority vote is more often correct. Similarly, the k -means algorithm with $d_{\mathbf{M}}$ builds Voronoi cells that are no longer balls but ellipsoids, and can thus recover elongated or correlated clusters: it was precisely to improve clustering with side information that one of the first metric learning algorithms was proposed [Xing et al. \(2002\)](#).

10.4.3 Learning a Metric from Side Information

To learn $\mathbf{M}$, we need some supervision telling us which examples should be close and which ones should be far apart. This supervision generally takes one of the following forms [Bellet et al. \(2013\)](#):

- **must-link** (or similarity) constraints: a set $\mathcal{S}$ of pairs $(\mathbf{x}_i, \mathbf{x}_j)$ such that $\mathbf{x}_i$ and $\mathbf{x}_j$ should be similar;
- **cannot-link** (or dissimilarity) constraints: a set $\mathcal{D}$ of pairs $(\mathbf{x}_i, \mathbf{x}_j)$ such that $\mathbf{x}_i$ and $\mathbf{x}_j$ should be dissimilar;
- **relative** (or triplet) constraints: a set $\mathcal{R}$ of triplets $(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k)$ such that $\mathbf{x}_i$ should be more similar to $\mathbf{x}_j$ than to $\mathbf{x}_k$.

We follow here the notation of [Bellet et al. \(2013\)](#): in this section only, $\mathcal{D}$ and $\mathcal{R}$ denote sets of constraints, and not the data distribution and the risk.

In supervised classification, these constraints are directly derived from the labels: two examples of the same class form a must-link pair, two examples of different classes

form a cannot-link pair, and a triplet $(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k)$ with $y_i = y_j \neq y_k$ forms a relative constraint. Such constraints are also available in weakly supervised settings where labels are unknown, *e.g.* when a user tells us that two images are “alike”. Metric learning then consists in solving a problem of the form

$$\min_{\mathbf{M} \in \mathbb{S}_+^d} \mathcal{L}(\mathbf{M}, \mathcal{S}, \mathcal{D}, \mathcal{R}) + \lambda \Omega(\mathbf{M}),$$

where the loss $\mathcal{L}$ penalizes the violated constraints and Ω is a regularizer (*e.g.* $\|\mathbf{M}\|_{\mathcal{F}}^2$, or the trace of $\mathbf{M}$ to favor low-rank solutions). A key observation is that the squared distance is **linear** in $\mathbf{M}$:

$$d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)^\top \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j) = \text{Tr}(\mathbf{M} \mathbf{C}_{ij}), \quad \text{with} \quad \mathbf{C}_{ij} = (\mathbf{x}_i - \mathbf{x}_j)(\mathbf{x}_i - \mathbf{x}_j)^\top.$$

Hence, constraints on squared distances are linear constraints on $\mathbf{M}$ and, together with the convex constraint $\mathbf{M} \succeq 0$, they often lead to convex optimization problems. Let us briefly present two classical algorithms.

MMC [Xing et al. \(2002\)](#) The first Mahalanobis metric learning method, sometimes called MMC, brings the similar pairs together while keeping the dissimilar pairs apart:

$$\min_{\mathbf{M} \in \mathbb{S}_+^d} \sum_{(\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{S}} d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) \quad \text{s.t.} \quad \sum_{(\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{D}} d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) \geq 1.$$

The constraint prevents the trivial solution $\mathbf{M} = \mathbf{0}$ (using $d_{\mathbf{M}}$ rather than $d_{\mathbf{M}}^2$ in the constraint avoids solutions of rank one). The problem is convex, and it is solved by a projected gradient method which requires an eigenvalue decomposition at each iteration, which becomes costly in high dimension. Note that MMC tries to bring *all* the similar pairs together, which is a strong requirement when a class is made of several separated groups.

ITML [Davis et al. \(2007\)](#) Information-Theoretic Metric Learning regularizes the learned matrix towards a prior matrix $\mathbf{M}_0$ (*e.g.* the identity):

$$\begin{aligned} \min_{\mathbf{M} \in \mathbb{S}_+^d} D_{ld}(\mathbf{M}, \mathbf{M}_0) \quad \text{s.t.} \quad & d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) \leq u \quad \forall (\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{S}, \\ & d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) \geq l \quad \forall (\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{D}, \end{aligned}$$

where u and l are two thresholds and $D_{ld}(\mathbf{M}, \mathbf{M}_0) = \text{Tr}(\mathbf{M} \mathbf{M}_0^{-1}) - \log \det(\mathbf{M} \mathbf{M}_0^{-1}) - d$ is the LogDet divergence, which is closely related to the Kullback-Leibler divergence between two Gaussian distributions of covariance matrices $\mathbf{M}^{-1}$ and $\mathbf{M}_0^{-1}$. This divergence is finite only for positive definite matrices, so that the iterates automatically remain in the PSD cone and no projection is needed. The problem is solved with Bregman projections, and slack variables are added when the constraints cannot all be satisfied.

10.4.4 Large Margin Nearest Neighbors (LMNN)

The Large Margin Nearest Neighbors algorithm (LMNN) of Weinberger and Saul [Weinberger and Saul \(2009\)](#) is one of the most widely used metric learning algorithms. It is specifically designed to improve the k -NN classifier, and its key idea is that, to classify $\mathbf{x}_i$ correctly, we do not need all the examples of its class to be close to $\mathbf{x}_i$ (as in MMC): it is enough that its k nearest neighbors belong to its class. The constraints are thus *local*.

Target neighbors and impostors Before learning, we associate with each training example $\mathbf{x}_i$ a set of k **target neighbors**: the k nearest examples (for the Euclidean distance, or using prior knowledge) that share the label of $\mathbf{x}_i$. We write $j \rightsquigarrow i$ when $\mathbf{x}_j$ is a target neighbor of $\mathbf{x}_i$. These target neighbors are fixed once and for all. We would like the target neighbors of $\mathbf{x}_i$ to be its nearest neighbors in the learned metric. The examples that prevent this are the **impostors**: an impostor of $\mathbf{x}_i$ is an example $\mathbf{x}_l$ with a different label, $y_l \neq y_i$, which lies inside the perimeter defined by the target neighbors of $\mathbf{x}_i$ enlarged by a margin, *i.e.* such that

$$d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_l) \leq d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) + 1 \quad \text{for some target neighbor } j \rightsquigarrow i.$$

The objective function LMNN combines two terms, illustrated in Figure [103](#). Denoting $y_{il} = 1$ if $y_i = y_l$ and $y_{il} = 0$ otherwise:

- a **pull** term, which brings the target neighbors closer:

$$\varepsilon_{\text{pull}}(\mathbf{M}) = \sum_{i=1}^m \sum_{j \rightsquigarrow i} d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j);$$

- a **push** term, which pushes the impostors out of the perimeter, with a margin of 1, using the hinge loss $[u]_+ = \max(0, u)$:

$$\varepsilon_{\text{push}}(\mathbf{M}) = \sum_{i=1}^m \sum_{j \rightsquigarrow i} \sum_{l=1}^m (1 - y_{il}) \left[1 + d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) - d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_l) \right]_+.$$

The learned matrix is then the solution of

$$\min_{\mathbf{M} \in \mathbb{S}_+^d} (1 - \mu) \varepsilon_{\text{pull}}(\mathbf{M}) + \mu \varepsilon_{\text{push}}(\mathbf{M}),$$

where $\mu \in [0, 1]$ balances the two terms ($\mu = 1/2$ usually works well). The unit margin fixes the scale of $\mathbf{M}$: without it, multiplying $\mathbf{M}$ by a small constant would make all the distances, and thus the pull term, arbitrarily small. The similarity with the soft-margin SVM (Section [8](#)) is striking, and justifies the name of the algorithm: a hinge loss enforces a margin, and a second term plays the role of a regularizer.

A semi-definite program Introducing a slack variable $\xi_{ijl} \geq 0$ for each triplet, as for the soft-margin SVM, and using $d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) = \text{Tr}(\mathbf{M}\mathbf{C}_{ij})$, the LMNN problem can

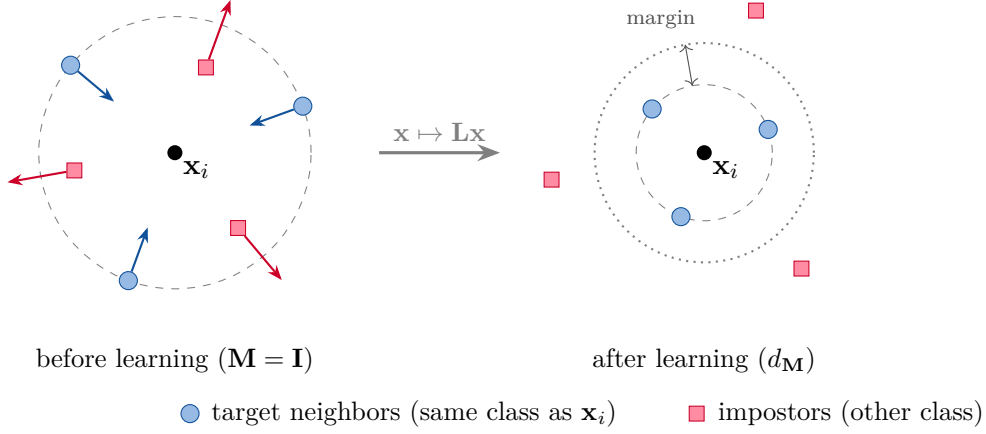


Figure 103: Principle of LMNN. Left: with the Euclidean distance, several examples of another class (impostors, red squares) are closer to $\mathbf{x}_i$ than its target neighbors (blue circles), so that the k -NN classifier makes an error. The pull term brings the target neighbors closer (blue arrows) and the push term moves the impostors away (red arrows). Right: in the learned metric, the target neighbors lie inside the perimeter of $\mathbf{x}_i$ (dashed circle) and the impostors lie beyond this perimeter plus a margin (dotted circle).

be rewritten as

$$\begin{aligned} \min_{\mathbf{M}, \xi} \quad & (1 - \mu) \sum_{i=1}^m \sum_{j \rightsquigarrow i} \text{Tr}(\mathbf{M} \mathbf{C}_{ij}) + \mu \sum_{i=1}^m \sum_{j \rightsquigarrow i} \sum_{l=1}^m (1 - y_{il}) \xi_{ijl} \\ \text{s.t.} \quad & \text{Tr}(\mathbf{M} \mathbf{C}_{il}) - \text{Tr}(\mathbf{M} \mathbf{C}_{ij}) \geq 1 - \xi_{ijl}, \quad \xi_{ijl} \geq 0, \quad \forall i, j \rightsquigarrow i, l : y_{il} = 0, \\ & \mathbf{M} \succeq 0. \end{aligned}$$

The objective and the constraints are linear in $(\mathbf{M}, \xi)$, and the variable $\mathbf{M}$ is constrained to belong to the cone of PSD matrices: this is a **semi-definite program** (SDP), a convex optimization problem. Consequently, LMNN does not suffer from local minima. However, generic SDP solvers do not scale well: the number of triplets is of the order of $k m^2$ and each iteration of an interior point method is expensive. Weinberger and Saul therefore proposed a dedicated first-order solver, which exploits the fact that only a small number of triplets are active (*i.e.* involve an actual impostor) at a given time.

Projected (sub)gradient descent Let us denote by $\varepsilon(\mathbf{M}) = (1 - \mu)\varepsilon_{\text{pull}}(\mathbf{M}) + \mu\varepsilon_{\text{push}}(\mathbf{M})$ the objective function. It is convex but not differentiable, because of the hinge loss. A subgradient of ε at $\mathbf{M}$ is given by

$$\mathbf{G}(\mathbf{M}) = (1 - \mu) \sum_{i=1}^m \sum_{j \rightsquigarrow i} \mathbf{C}_{ij} + \mu \sum_{(i,j,l) \in \mathcal{A}(\mathbf{M})} (\mathbf{C}_{ij} - \mathbf{C}_{il}),$$

where $\mathcal{A}(\mathbf{M})$ is the set of *active* triplets, *i.e.* the triplets (i, j, l) with $j \rightsquigarrow i$, $y_{il} = 0$ and $1 + d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) - d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_l) > 0$. A gradient step $\mathbf{M} - \eta \mathbf{G}(\mathbf{M})$ may leave the cone $\mathbb{S}_+^d$; we then project it back onto the cone, which is easy thanks to the following result.

Algorithm 11: LMNN by projected subgradient descent

Inputs : Training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, number of target neighbors k , trade-off $\mu \in [0, 1]$, step size $\eta > 0$, number of iterations T

Output: A matrix $\mathbf{M} \in \mathbb{S}_+^d$

```
1 for  $i = 1$  to  $m$  do
2   find the  $k$  target neighbors  $j \rightsquigarrow i$  of  $\mathbf{x}_i$ : its  $k$  nearest (Euclidean) neighbors
   with label  $y_i$ ;
3 set  $\mathbf{M}_0 = \mathbf{I}$ ;
4 for  $t = 0$  to  $T - 1$  do
5   compute the set  $\mathcal{A}(\mathbf{M}_t)$  of active triplets (target neighbors violated by an
   impostor);
6   compute the subgradient
    $\mathbf{G}(\mathbf{M}_t) = (1 - \mu) \sum_i \sum_{j \rightsquigarrow i} \mathbf{C}_{ij} + \mu \sum_{(i,j,l) \in \mathcal{A}(\mathbf{M}_t)} (\mathbf{C}_{ij} - \mathbf{C}_{il})$ ;
7   gradient step:  $\mathbf{A} = \mathbf{M}_t - \eta \mathbf{G}(\mathbf{M}_t)$ ;
8   projection: compute  $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$  and set  $\mathbf{M}_{t+1} = \mathbf{U} \max(\mathbf{\Lambda}, 0) \mathbf{U}^\top$ ;
9 return  $\mathbf{M}_T$ 
```

Proposition 10.5: Projection onto the PSD cone

Let $\mathbf{A}$ be a symmetric matrix with eigendecomposition $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$, where $\mathbf{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_d)$. Then the projection of $\mathbf{A}$ onto $\mathbb{S}_+^d$ for the Frobenius norm is

$$\Pi_{\mathbb{S}_+^d}(\mathbf{A}) = \arg \min_{\mathbf{M} \in \mathbb{S}_+^d} \|\mathbf{M} - \mathbf{A}\|_{\mathcal{F}} = \mathbf{U} \max(\mathbf{\Lambda}, 0) \mathbf{U}^\top,$$

where $\max(\mathbf{\Lambda}, 0) = \text{diag}(\max(\lambda_1, 0), \dots, \max(\lambda_d, 0))$, *i.e.* the negative eigenvalues are set to 0.

Proof. The Frobenius norm is invariant under orthogonal transformations, so $\|\mathbf{M} - \mathbf{A}\|_{\mathcal{F}} = \|\mathbf{B} - \mathbf{\Lambda}\|_{\mathcal{F}}$ with $\mathbf{B} = \mathbf{U}^\top \mathbf{M} \mathbf{U}$, and $\mathbf{B} \succeq 0$ if and only if $\mathbf{M} \succeq 0$. For any $\mathbf{B} \succeq 0$, the diagonal entries satisfy $B_{kk} = \mathbf{e}_k^\top \mathbf{B} \mathbf{e}_k \geq 0$, hence

$$\|\mathbf{B} - \mathbf{\Lambda}\|_{\mathcal{F}}^2 = \sum_{k=1}^d (B_{kk} - \lambda_k)^2 + \sum_{k \neq k'}^d B_{kk'}^2 \geq \sum_{k=1}^d (B_{kk} - \lambda_k)^2 \geq \sum_{k=1}^d \min(\lambda_k, 0)^2.$$

This lower bound is attained by $\mathbf{B} = \max(\mathbf{\Lambda}, 0)$, which is PSD, *i.e.* by $\mathbf{M} = \mathbf{U} \max(\mathbf{\Lambda}, 0) \mathbf{U}^\top$. $\square$

The resulting procedure is summarized in Algorithm 11. Each iteration requires an eigenvalue decomposition, whose cost is $O(d^3)$: this is affordable for moderate dimensions, and a preliminary dimensionality reduction (*e.g.* with a principal component analysis) is often applied beforehand. An alternative, used in many implementations, is to optimize directly over the linear map $\mathbf{L}$ of Proposition 10.4: the PSD constraint then disappears, but the problem is no longer convex in $\mathbf{L}$.

Once $\mathbf{M}$ is learned, a new example is classified with the k -NN rule using $d_{\mathbf{M}}$ or, equivalently, with the Euclidean k -NN rule applied to the transformed data $\mathbf{L}\mathbf{x}$. On many benchmarks, LMNN significantly improves the accuracy of the k -NN classifier compared with the Euclidean distance [Weinberger and Saul \(2009\)](#).

Going further We stop our presentation of metric learning here. Let us simply mention that many other approaches exist, *e.g.* metric learning with generalization guarantees, local metric learning, which learns several metrics in different regions of the space [Zantedeschi et al. \(2016\)](#), or nonlinear metric learning, where the linear map $\mathbf{L}$ is replaced by a neural network trained on pairs or triplets of examples (siamese networks [Bromley et al. \(1994\)](#); [Chopra et al. \(2005\)](#)). We refer the interested reader to the survey [Bellet et al. \(2013\)](#) and to the book [Bellet et al. \(2015\)](#) for a comprehensive overview.

To remember from Section 10

- The expected squared error decomposes into $\text{bias}^2 + \text{variance} + \text{noise}$; more generally, the excess risk is the sum of an approximation error and of an estimation error, the latter being controlled by the Rademacher complexity of $\mathcal{H}$ (Theorem 10.2).
- Averaging T models never increases the squared error, and divides the variance by T when the errors are uncorrelated; correlation between the models limits the gain.
- The three ensemble strategies of this section are compared in the table below.
- AdaBoost is a coordinate descent on the exponential loss; its training error is at most $\exp(-2\gamma^2 T)$ when every weak learner has an edge γ .
- Gradient boosting is a gradient descent in function space: each weak learner fits the negative gradient (pseudo-residuals) of any differentiable loss; XGBoost uses a second-order expansion to compute the leaf values and the splitting gain. Use a small learning rate and early stopping.
- Stacking learns how to combine models; its meta-model must be trained on out-of-fold predictions.
- A Mahalanobis distance $d_{\mathbf{M}}$ is a Euclidean distance after the linear map $\mathbf{L}$, with $\mathbf{M} = \mathbf{L}^\top \mathbf{L}$; LMNN learns it with local pull/push constraints to improve the k -NN classifier.

	bagging, random forests	boosting	stacking
base learners	strong (deep trees)	weak (stumps, small trees)	any, heterogeneous
training	parallel, bootstrap	sequential, reweighting	base models, then meta-model
mainly reduces	the variance	the bias (and the variance)	both
large T	no over-fitting	over-fitting: early stopping	–

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

Back to Statistical Learning Theory

- Q10.1.** Data follow the model $y = f(\mathbf{x}) + \varepsilon$ with $\text{Var}(\varepsilon) = 0.25$. At a point $\mathbf{x}$, a learned predictor satisfies $\mathbb{E}[h(\mathbf{x})] = f(\mathbf{x}) + 0.3$ and $\text{Var}(h(\mathbf{x})) = 0.16$. What is its expected squared error at $\mathbf{x}$, and which part of it could a better algorithm remove?
- Q10.2.** In the decomposition of the excess risk $\mathcal{R}(h) - \mathcal{R}(h_{\mathcal{H}})$, why can the term $\mathcal{R}_S(h_{\mathcal{H}}) - \mathcal{R}(h_{\mathcal{H}})$ be controlled by the law of large numbers (or by Hoeffding's inequality), whereas the term $\mathcal{R}(h) - \mathcal{R}_S(h)$ cannot?
- Q10.3.** Assume that $\mathfrak{R}_m(\mathcal{H}) \leq C/\sqrt{m}$ for some constant C . By how much is the gap $\mathcal{R}(h) - \mathcal{R}_S(h)$ guaranteed by the first bound of Theorem 10.2 reduced when the sample size is multiplied by 4?

Ensemble Methods

- Q10.4.** A model has a very small training error and a large test error. Which term of the error decomposition is large, and which ensemble method would you try first?
- Q10.5.** What fraction of the distinct examples of S appears, on average, in a bootstrap sample of size m , when m is large?
- Q10.6.** Why does a random forest draw a random subset of the features at each node, rather than just using bootstrap samples?
- Q10.7.** True or false? “A random forest with 1000 trees over-fits more than the same forest with 100 trees.”

Ensemble Methods

- Q10.8.** In AdaBoost, what happens to the weight α_t when $\varepsilon_t = 1/2$? And to the weights of the examples misclassified by h_t when $\varepsilon_t < 1/2$?
- Q10.9.** The weak learner of AdaBoost returns a classifier h_t with a weighted error $\varepsilon_t = 0.8$. Is it useless? What does AdaBoost do with it?
- Q10.10.** Why should the meta-model of a stacking procedure not be trained on the predictions of the base learners on their own training data?

Gradient Boosting

- Q10.11.** What are the pseudo-residuals of gradient boosting for the loss $\frac{1}{2}(y - H)^2$? Does increasing the number of trees always improve a gradient boosting model?
- Q10.12.** With the logistic loss and labels $y \in \{0, 1\}$, the pseudo-residual is $y - \sigma(H_{t-1}(\mathbf{x}))$. Compute it for a positive example ($y = 1$) predicted with probability 0.9, then with probability 0.1. Which one will the next weak learner focus on?
- Q10.13.** In XGBoost with the squared loss $\frac{1}{2}(y - \hat{y})^2$, a leaf contains three examples whose residuals $y_i - \hat{y}_i^{(t-1)}$ are 1, 2 and 3. Compute the value of the leaf for $\lambda = 0$ and for $\lambda = 1$.

Synthesis of the section

- Q10.14.** Let $\mathbf{M} = \text{diag}(1, 0)$. Is $d_{\mathbf{M}}$ a distance on $\mathbb{R}^2$? Give a matrix $\mathbf{L}$ such that $\mathbf{M} = \mathbf{L}^\top \mathbf{L}$.
- Q10.15.** True or false? “Standardizing the features is enough to make the Euclidean k -NN classifier insensitive to irrelevant features.”
- Q10.16.** LMNN can be solved either over the matrix $\mathbf{M} \succeq 0$ or directly over the linear map $\mathbf{L}$, with $\mathbf{M} = \mathbf{L}^\top \mathbf{L}$. What is gained and what is lost in the second case?

11 Applications and Learning in Practice

The previous sections presented many algorithms together with their theoretical properties. In practice, however, choosing the “best” algorithm is rarely the most difficult part of a machine learning project: most of the time is spent understanding the problem, collecting and cleaning the data, designing a sound evaluation protocol and interpreting the results. A powerful model evaluated with a flawed protocol is worse than useless, since it gives a false sense of confidence.

This section is a practical guide to conduct a supervised learning study, from the formulation of the problem to the report of the results. It gathers advice that is scattered throughout the literature and that is usually learned the hard way. We illustrate some of the steps with short pieces of `Python` code using the `scikit-learn` library [Pedregosa et al. \(2011\)](#), but the principles do not depend on the software. As a running illustration, the reader may think of a fraud detection problem: predicting whether a banking transaction is fraudulent from its characteristics [Dal Pozzolo et al. \(2015\)](#); [Metzler et al. \(2018b\)](#).

About this section

In this section you will learn to:

- *frame a prediction problem and choose a performance measure before looking at the results;*
- *explore and preprocess the data (missing values, categorical features, scaling) without leaking information from the test set;*
- *split the data according to their structure (stratified, grouped or temporal) and evaluate a whole tuning procedure by nested cross-validation;*
- *handle imbalanced data with adapted measures, resampling, costs and decision thresholds;*
- *interpret a model, report its performance with its uncertainty and compare models with appropriate statistical tests.*

11.1 Framing the Problem and Choosing a Metric

Before any computation, the problem must be translated into a well-defined learning task. This requires answering a few questions with the people who will use the model.

- **What exactly do we predict, and for which unit?** A transaction, a customer, a day? Is it a classification task (fraudulent or not) or a regression task (amount of the loss)? Is the label reliable, and when is it known (a fraud may only be detected weeks later)?
- **Which information is available at prediction time?** A feature that is only known *after* the event we want to predict must not be used, even if it is present in the historical database (see [Section 11.4](#)).
- **What is the cost of the errors?** Missing a fraud (a false negative) and blocking a legitimate transaction (a false positive) do not have the same cost. These costs determine the performance measure and, as we will see, the decision threshold of the model.

- **What is a useful level of performance?** Comparing the model with the current solution (*e.g.* a set of expert rules) or with a trivial model tells us whether machine learning brings any value.

The choice of the **performance measure** follows from these answers, and it must be made *before* looking at the results, to avoid choosing the measure that makes our favorite model look good. The classical measures have been presented in Section 7.2: the accuracy is only meaningful when the classes are balanced and the errors have similar costs; otherwise, the precision, the recall, the F -measure, the area under the ROC or precision-recall curves, or an explicit cost are more appropriate. In regression, the choice between the mean squared error and the mean absolute error depends on how much large errors should be penalized.

11.2 Collecting and Exploring the Data

Machine learning assumes that the training data are representative of the data the model will face in production, *i.e.* that both are drawn from the same distribution $\mathcal{D}$. This assumption must be questioned from the start: was the sample collected in a biased way (*e.g.* only the transactions that were manually checked have a label)? Did the distribution change over time (new fraud patterns, a new version of a sensor)?

An **exploratory data analysis** then aims at understanding the data before modeling them. It typically includes:

- the size of the dataset and the type of each feature (numerical, categorical, ordinal, date, text), together with a description of what each feature means;
- the distribution of the target: proportion of each class, or histogram of the target value (a strong imbalance calls for the methods of Section 11.6);
- univariate summaries (mean, median, quantiles, number of distinct values) and plots of each feature, which reveal errors of encoding, impossible values or strong asymmetries;
- the proportion and the structure of the missing values, and the presence of duplicated examples;
- the relations between the features (correlation matrix, scatter plots) and between each feature and the target;
- the temporal or grouped structure of the data (several examples per customer, per patient, per day), which will matter when splitting the data.

Suspicious findings, such as a feature that predicts the target almost perfectly, must be investigated: they are more often the sign of a problem (*e.g.* a leakage) than of a miracle.

11.3 Preprocessing and Feature Engineering

Most algorithms expect a numerical matrix without missing values. Preprocessing transforms the raw data into such a matrix, and feature engineering tries to build a representation in which the task is easier to learn.

Missing values Removing the examples (or the features) with missing values is the simplest solution, but it wastes data and may bias the sample when the values are not missing at random (*e.g.* people with a high income may be less likely to report it). The most common alternative is *imputation*: replacing a missing value by the mean or the median of the feature (for numerical features), by its most frequent value (for categorical features), or by a value predicted from the other features (*e.g.* with a k -NN or a regression model). Since the fact that a value is missing can itself be informative, it is often useful to add a binary *missingness indicator* feature. Note that some algorithms, such as several implementations of gradient boosting, handle missing values natively.

Categorical features A categorical feature with q modalities is usually encoded with a *one-hot encoding*: it is replaced by q binary features, exactly one of which is equal to 1. Encoding the modalities by integers $1, \dots, q$ is only appropriate for ordinal features (*e.g.* a level of education), since it introduces an order and distances between the modalities. When q is very large (*e.g.* a zip code or a merchant identifier), one can group the rare modalities, or use a *target encoding*, which replaces each modality by the mean of the target over the examples having this modality; the latter must be computed on the training data only (and, ideally, out-of-fold), otherwise it leaks the label into the features.

Scaling Algorithms based on distances (k -NN, k -means, kernel methods), on gradient descent (neural networks) or on a penalty on the weights (ridge, lasso, SVM) are sensitive to the scale of the features. Decision trees and their ensembles are not, since their splits only depend on the order of the values.

Definition 11.1: Standardization

Let $\{\mathbf{x}_i\}_{i=1}^m$ be a training sample in $\mathbb{R}^d$. The standardization of a feature j transforms any value x_j (of a training or a new example) into

$$\tilde{x}_j = \frac{x_j - \hat{\mu}_j}{\hat{s}_j}, \quad \text{where} \quad \hat{\mu}_j = \frac{1}{m} \sum_{i=1}^m x_{ij} \quad \text{and} \quad \hat{s}_j^2 = \frac{1}{m} \sum_{i=1}^m (x_{ij} - \hat{\mu}_j)^2$$

are estimated on the **training** sample only.

The standardized features have zero mean and unit variance on the training set. Another common choice is the *min-max* scaling, which maps each feature to $[0, 1]$ using its minimum and maximum on the training set. In the vocabulary of Section 10.4, standardizing the data before a k -NN is the same as using a Mahalanobis distance with a diagonal matrix $\mathbf{M}$ of entries $1/\hat{s}_j^2$; metric learning goes further by learning $\mathbf{M}$ from the labels.

Outliers Outliers can be detected with simple univariate rules (*e.g.* a value more than 1.5 interquartile ranges below the first quartile or above the third quartile) or with dedicated algorithms Breunig et al. (2000). They should not be removed blindly: an outlier may be an error of measurement, but it may also be the most interesting example of the dataset (a fraud is, by nature, an unusual transaction). Robust alternatives

consist in transforming the feature (*e.g.* with a logarithm), clipping extreme values, or using robust estimators (median and interquartile range instead of mean and standard deviation).

Feature engineering Knowledge of the domain is often the most effective way to improve a model. Typical transformations include applying a logarithm to a skewed positive feature (amounts, counts), extracting information from dates (hour, day of the week, time since the last event), computing ratios or interactions between features, and aggregating the history of an entity (*e.g.* the number of transactions of the card in the last hour, which is known to be very informative for fraud detection). Conversely, *feature selection* removes irrelevant or redundant features, which reduces the variance of the model and its cost. As always, every transformation whose parameters are estimated from the data must be estimated on the training set only.

11.4 Splitting the Data and Avoiding Leakage

The protocol of Section 7 splits the data into a training set, used to learn and tune the model (possibly with a cross-validation, see Figure 49), and a test set, used once at the end to estimate the generalization performance. This protocol is only valid if the test set mimics new, unseen data. The way the data are split must therefore respect their structure:

- in classification, the split is usually *stratified*, so that each set has the same class proportions (this is essential with imbalanced data);
- when several examples come from the same entity (a customer, a patient), all of them must belong to the same set (*group split*), otherwise the model is evaluated on entities it has already seen;
- when the data are ordered in time and the model will predict the future, the training set must precede the test set (*temporal split*); a random split would let the model use the future to predict the past.

Definition 11.2: Data leakage

There is a *data leakage* when information that would not be available at prediction time, in particular information about the test examples or about the target, is used to train or to select a model [Kaufman et al. \(2012\)](#).

A leakage leads to an optimistic estimate of the performance, which collapses once the model is deployed. Classical sources of leakage are:

- features that are consequences of the target (*e.g.* a “refund issued” flag to predict a fraud, or a treatment to predict a diagnosis);
- preprocessing parameters (means, variances, imputation values, selected features, target encodings) estimated on the whole dataset before splitting it;
- duplicated (or near-duplicated) examples present in both the training and the test sets, or a random split of grouped or temporal data;

- using the test set, even indirectly, to choose the model, the features or the hyper-parameters.

The simplest way to avoid the second kind of leakage is to gather all the steps of preprocessing and learning into a single object, a *pipeline*, which is fitted on the training data only and then applied, unchanged, to the test data. When a pipeline is evaluated by cross-validation, the preprocessing is re-fitted on the learning folds at each round, so that the validation fold never influences it.

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

num = Pipeline([("impute", SimpleImputer(strategy="median")),
                 ("scale", StandardScaler())])
cat = Pipeline([("impute", SimpleImputer(strategy="most_frequent")),
                 ("onehot", OneHotEncoder(handle_unknown="ignore"))])
prep = ColumnTransformer([("num", num, num_cols),
                           ("cat", cat, cat_cols)])
model = Pipeline([("prep", prep),
                  ("clf", LogisticRegression(max_iter=1000))])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=0)
# imputation and scaling are fitted on X_train only
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

11.5 Choosing, Tuning and Comparing Models

Start with baselines Any study should start with *baselines*: a trivial model (predicting the majority class, or the mean of the target in regression) and a simple, well-understood model (a regularized linear or logistic regression, a small decision tree). They provide a reference: a complex model is only worth its cost if it clearly outperforms them. They also help detect bugs, *e.g.* a model that does not beat the trivial one on the training set. Then, more flexible models can be tried; for tabular data, ensembles of trees such as random forests and gradient boosting (Sections 10.2.1 and 10.3) are often very strong candidates.

Diagnosing a model Comparing the training and validation errors tells us in which regime we are (Section 10): a high training error indicates a large bias (under-fitting), and calls for a more flexible model or better features; a low training error with a much higher validation error indicates a large variance (over-fitting), and calls for more data, more regularization or an ensemble method. Plotting these errors as a function of the size of the training set (*learning curves*) tells us whether collecting more data would help.

Hyper-parameter search The hyper-parameters are chosen by cross-validation, as explained in Section 7. The candidate values can be explored in two main ways.

- **Grid search** evaluates all the combinations of a finite set of values for each hyper-parameter. Its cost grows exponentially with the number of hyper-parameters.
- **Random search** evaluates a fixed number of combinations drawn at random from given distributions (*e.g.* a log-uniform distribution for a regularization constant). For the same budget, it is often more efficient than a grid search [Bergstra and Bengio \(2012\)](#): when only a few hyper-parameters really matter, a grid of n^2 points only tests n distinct values of each of them, whereas n^2 random points test n^2 distinct values of each of them.

More sophisticated strategies, such as Bayesian optimization, build a model of the validation score as a function of the hyper-parameters to choose the next candidates, and are useful when each evaluation is expensive.

Nested cross-validation A subtle but important point is that the best cross-validation score obtained during the search is *not* an unbiased estimate of the performance of the selected model: by selecting the maximum of many noisy scores, we select, in part, the noise [Cawley and Talbot \(2010\)](#); [Varma and Simon \(2006\)](#).

Proposition 11.1: Optimism of the selected score

Let $\hat{s}_1, \dots, \hat{s}_K$ be unbiased estimates of the true performances $s_1, \dots, s_K$ of K candidate models, *i.e.* $\mathbb{E}[\hat{s}_k] = s_k$ for all k . Then

$$\mathbb{E} \left[\max_k \hat{s}_k \right] \geq \max_k s_k.$$

Proof. For all k , we have $\max_{k'} \hat{s}_{k'} \geq \hat{s}_k$, hence $\mathbb{E}[\max_{k'} \hat{s}_{k'}] \geq \mathbb{E}[\hat{s}_k] = s_k$. Taking the maximum over k of the right-hand side gives the result. $\square$

The inequality is usually strict, and the bias increases with the number of candidates and with the variance of the estimates (small datasets). This is why the final performance must be measured on data that played no role in the selection: either a separate test set, or, when data are scarce, a **nested cross-validation**. In a nested cross-validation, an *outer* cross-validation estimates the performance, and, within each outer training set, an *inner* cross-validation selects the hyper-parameters (Figure 104). The outer validation folds are thus never used for the selection.

Note that nested cross-validation evaluates the *whole procedure* (search included); the final model is then obtained by running the search on all the data.

```
from sklearn.model_selection import (GridSearchCV, StratifiedKFold,
                                     cross_val_score)
inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
outer = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
search = GridSearchCV(model, {"clf__C": [0.01, 0.1, 1, 10, 100]},
                      cv=inner, scoring="f1")
```

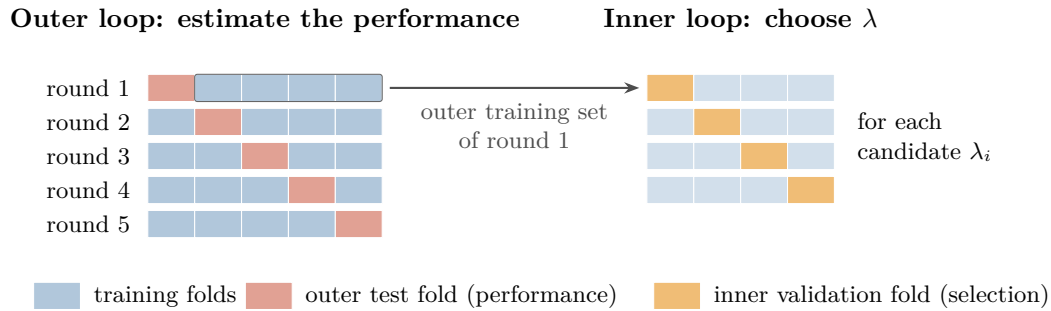


Figure 104: Nested 5×4 -fold cross-validation. In each outer round, the four blue folds form the outer training set: an inner 4-fold cross-validation run on them chooses the hyper-parameter λ , a model is refitted on them with this value, and it is scored on the red outer test fold. The red folds never take part in the selection, so the average of the five outer scores estimates the performance of the whole procedure (search included).

```
scores = cross_val_score(search, X_train, y_train, cv=outer,
                          scoring="f1")
print(scores.mean(), scores.std())
```

The same reasoning applies to ensembles: the base learners of a stacking model (Section 10.2.3) can be tuned in an inner loop, while the meta-model is learned on out-of-fold predictions.

11.6 Learning from Imbalanced Data

In many applications, such as fraud detection, medical diagnosis or anomaly detection, the class of interest (the *positive* or *minority* class) is much rarer than the other one: the *imbalance ratio* between the majority and the minority classes can reach 100 or 1000. In this case, a classifier that always predicts the majority class has an excellent accuracy but is useless, and most learning algorithms, which minimize a surrogate of the error rate, tend to neglect the minority class Lopez et al. (2013). Learning from imbalanced data has been an active research topic for more than twenty years, to which the author of these notes has contributed Metzler et al. (2018a,b); Viola et al. (2019); Bascol et al. (2019). The approaches can be grouped into three families.

Adapted performance measures The first step is to use a measure that focuses on the minority class (Section 7.2): the recall, the precision, the F -measure, the area under the precision-recall curve, or the balanced accuracy (the mean of the recalls of the two classes). The area under the ROC curve Metz (1978); Cortes and Mohri (2004) is insensitive to the imbalance, which can make it over-optimistic when the positive class is very rare.

Data-level methods: resampling These methods modify the training set to balance the classes, and can be used with any algorithm Batista et al. (2003). *Random under-sampling* removes examples of the majority class; it is fast but discards information. *Random oversampling* duplicates examples of the minority class; it keeps all the information but tends to over-fit the duplicated points. The SMOTE algorithm Chawla et al.

(2002) (Synthetic Minority Over-sampling TEchnique) generates instead *new* synthetic minority examples: for a minority example $\mathbf{x}_i$, it draws one of its k nearest minority neighbors $\mathbf{x}_{nn}$ and creates

$$\mathbf{x}_{\text{new}} = \mathbf{x}_i + u(\mathbf{x}_{nn} - \mathbf{x}_i), \quad \text{with } u \sim \mathcal{U}([0, 1]),$$

i.e. a random point on the segment joining $\mathbf{x}_i$ and $\mathbf{x}_{nn}$ (Figure 105). Many variants exist, focusing on the examples close to the decision boundary (Borderline-SMOTE Han et al. (2005)) or on the hardest examples (ADASYN He et al. (2008)), or combining oversampling with the removal of ambiguous pairs of examples (Tomek links Tomek (1976)); see Fernández et al. (2018) for a review. These methods are implemented in the `imbalanced-learn` library Lemaître et al. (2017). Resampling can also be combined with ensemble methods, as in SMOTEBoost Chawla et al. (2003); see Galar et al. (2012) for a review.

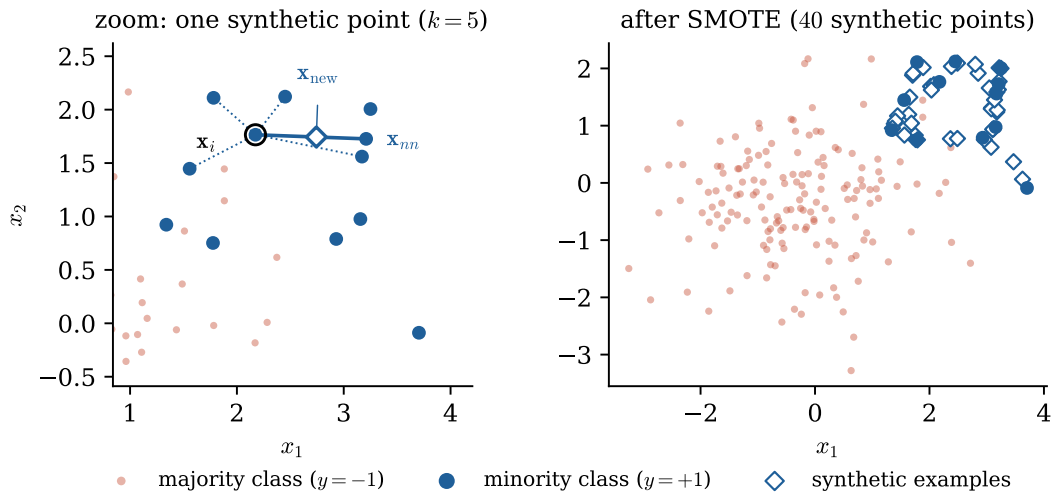


Figure 105: The SMOTE algorithm. Left: a synthetic example $\mathbf{x}_{\text{new}}$ is drawn at random on the segment joining a minority example $\mathbf{x}_i$ (circled) and one of its $k = 5$ nearest minority neighbors (dotted segments), here $\mathbf{x}_{nn}$. Right: 40 synthetic examples generated in this way; they fill the region spanned by the minority class, but never leave it.

[code: `make_figures_ch06.py`:
`fig_ch10_smote`]

Pitfall Resampling must only be applied to the **training** data (and, within a cross-validation, only to the learning folds). Oversampling before splitting puts copies, or interpolations, of the same minority examples in both the training and the validation sets: this is a leakage, which leads to very optimistic estimates. The validation and test sets must keep the true class proportions.

```
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
model = ImbPipeline([("prep", prep),
                      ("smote", SMOTE(k_neighbors=5)),
                      ("clf", RandomForestClassifier())])
# the sampler is only applied when fitting (learning folds)
```

```
scores = cross_val_score(model, X_train, y_train, cv=outer,
                          scoring="f1")
```

Algorithm-level methods: costs and thresholds Instead of modifying the data, we can modify the learning algorithm so that errors on the minority class cost more. This is the principle of *cost-sensitive learning* Elkan (2001); López et al. (2012). The following result, which follows directly from the definition of the Bayes classifier, shows that costs essentially move the decision threshold.

Proposition 11.2: Optimal decision with asymmetric costs Elkan (2001)

Let $\eta(\mathbf{x}) = \mathbb{P}(Y = 1 \mid \mathbf{x})$, and let $c_{FN} > 0$ (resp. $c_{FP} > 0$) be the cost of predicting 0 for a positive example (resp. 1 for a negative example), correct predictions having no cost. Then the classifier minimizing the expected cost predicts 1 if and only if

$$\eta(\mathbf{x}) \geq \frac{c_{FP}}{c_{FP} + c_{FN}}.$$

Proof. At a given $\mathbf{x}$, predicting 1 has an expected cost $(1 - \eta(\mathbf{x})) c_{FP}$, while predicting 0 has an expected cost $\eta(\mathbf{x}) c_{FN}$. Predicting 1 is optimal if and only if $(1 - \eta(\mathbf{x})) c_{FP} \leq \eta(\mathbf{x}) c_{FN}$, i.e. $\eta(\mathbf{x}) (c_{FP} + c_{FN}) \geq c_{FP}$. $\square$

With equal costs, we recover the usual threshold $1/2$. With $c_{FN} = 9 c_{FP}$, a transaction should be flagged as soon as its probability of being fraudulent exceeds 0.1 (Figure 106).

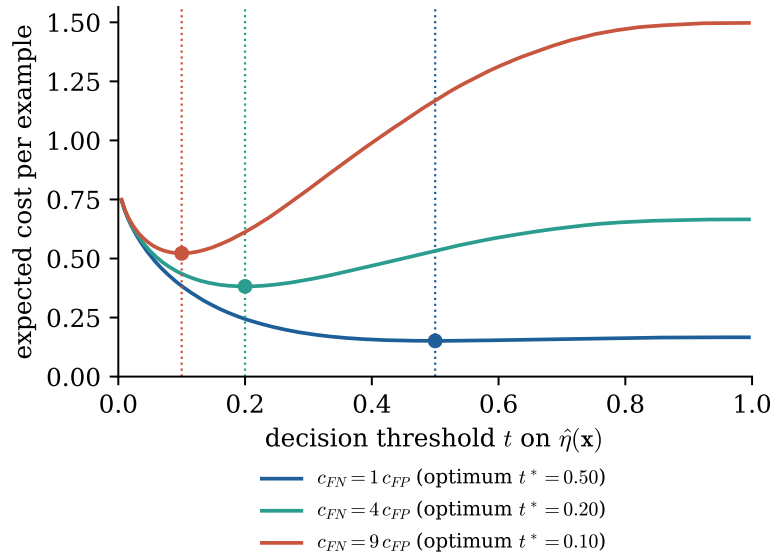


Figure 106: Expected cost per example of the classifier “predict 1 if $\eta(\mathbf{x}) \geq t$ ” as a function of the threshold t , for $c_{FP} = 1$ and three values of c_{FN} (the probabilities $\eta(\mathbf{x})$ follow a Beta(0.6, 3) distribution, i.e. about 17% of positives). The minimum is reached at $t^* = c_{FP}/(c_{FP} + c_{FN})$, as stated by Proposition 11.2: the more costly the false negatives, the lower the threshold.

[code: make_figures_ch06.py: fig_ch10_cost_threshold]

In practice, costs can be taken into account in three ways: by weighting the examples of each class in the loss function (the `class_weight` option of many `scikit-learn` models), by learning a model that estimates $\eta(\mathbf{x})$ and then moving the threshold, or by designing cost-sensitive versions of the algorithms, *e.g.* of decision trees [Ling et al. \(2006\)](#); [Metzler et al. \(2018b\)](#) or boosting [Fan et al. \(1999\)](#). Moving the threshold requires reasonably calibrated probabilities; note that resampling distorts the predicted probabilities, since it changes the class proportions seen during training.

When the target measure is not a cost but, *e.g.*, the F -measure, the optimal classifier is still a threshold on $\eta(\mathbf{x})$, but the optimal threshold depends on the distribution [Koyejo et al. \(2014\)](#); [Narasimhan et al. \(2015\)](#). The threshold (or, equivalently, the costs) is then tuned on validation data, and the F -measure can be optimized through a sequence of cost-sensitive problems [Parambath et al. \(2014\)](#); [Bascol et al. \(2019\)](#). In all cases, the threshold is a hyper-parameter and must be tuned without using the test set.

11.7 Interpreting a Model

A model with a good performance is not always sufficient: users, regulators or domain experts want to understand *why* it makes a given decision, and interpretation tools are also very effective to detect leakages or spurious correlations. We distinguish *global* explanations, which describe the behavior of the model as a whole (which features matter?), from *local* explanations, which explain a single prediction (why was this transaction flagged?).

Interpretable models Some models are interpretable by design: the coefficients of a linear or logistic regression (provided the features are standardized, so that the coefficients are comparable), or a small decision tree, whose decision can be read as a set of rules. For ensembles, the impurity-based importance of the features of a random forest (Section [10.2.1](#)) gives a first global view, but it is computed on the training data and it favors features with many possible split values.

Permutation importance A model-agnostic alternative, introduced with random forests [Breiman \(2001\)](#), consists in measuring how much the performance of the model degrades when the information carried by a feature is destroyed.

Definition 11.3: Permutation importance

Let h be a trained model, s a performance measure and V a validation (or test) set. For a feature j and a random permutation π of the examples of V , let $V^{(j,\pi)}$ be the set V in which the values of the j -th feature have been permuted according to π (the other features and the labels being unchanged). The permutation importance of the feature j is

$$I_j = s(h, V) - \frac{1}{R} \sum_{r=1}^R s(h, V^{(j,\pi_r)}),$$

where $\pi_1, \dots, \pi_R$ are R independent random permutations.

Permuting a feature breaks its relation with the target while keeping its marginal distribution. A large drop of performance means that the model relies heavily on the feature. The measure is computed on held-out data and applies to any model (`sklearn.inspection.permutation_importance`). Its main limitation concerns correlated features: when two features carry the same information, permuting one of them barely changes the performance, and both may appear unimportant.

SHAP values To explain an individual prediction $h(\mathbf{x})$, the SHAP method [Lundberg and Lee \(2017\)](#) borrows the Shapley value from cooperative game theory: the features are seen as players who cooperate to produce the prediction, and each feature receives its fair share of the “gain” $h(\mathbf{x}) - \mathbb{E}[h(\mathbf{X})]$. Denoting by $F = \{1, \dots, d\}$ the set of features and by $v_{\mathbf{x}}(A)$ the expected prediction when only the features of $A \subseteq F$ are known (e.g. $v_{\mathbf{x}}(A) = \mathbb{E}[h(\mathbf{X}) \mid \mathbf{X}_A = \mathbf{x}_A]$), the contribution of the feature j is

$$\phi_j(\mathbf{x}) = \sum_{A \subseteq F \setminus \{j\}} \frac{|A|! (d - |A| - 1)!}{d!} (v_{\mathbf{x}}(A \cup \{j\}) - v_{\mathbf{x}}(A)),$$

i.e. the average, over all the orders in which the features can be added, of the marginal contribution of j . These contributions are additive: $h(\mathbf{x}) = \phi_0 + \sum_{j=1}^d \phi_j(\mathbf{x})$, with $\phi_0 = \mathbb{E}[h(\mathbf{X})]$. The exact computation is exponential in d , so SHAP relies on approximations, and on an exact polynomial algorithm for tree ensembles. Averaging $|\phi_j(\mathbf{x})|$ over a dataset also gives a global importance measure.

Remark Explanations describe the *model*, not the world: a feature can be important for the model because it is correlated with the true cause, and an explanation does not imply causality. Moreover, different explanation methods may disagree, especially in the presence of correlated features.

11.8 Reproducibility and Reporting

A result that cannot be reproduced cannot be trusted. Reproducibility requires fixing the random seeds (of the splits, of the algorithms, of the resampling), recording the versions of the software and of the data, keeping the code of the whole pipeline (from the raw data to the figures), and saving the splits that were used.

Uncertainty of the results A performance measured on a finite test set is a random quantity, and it should always be reported with an estimate of its uncertainty: the standard deviation over the folds of a cross-validation or over several runs with different seeds, or a confidence interval. For the accuracy $\hat{p}$ measured on n independent test examples, the normal approximation of the binomial distribution gives the 95% confidence interval

$$\hat{p} \pm 1.96 \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}.$$

Example 11.1. A model correctly classifies 850 of the $n = 1000$ examples of a test set, so $\hat{p} = 0.85$ and $\sqrt{0.85 \times 0.15/1000} \simeq 0.0113$. The 95% confidence interval is approximately $[0.828, 0.872]$. A second model reaching 0.86 on the same test set is therefore not obviously better: the difference is well within the uncertainty of each

estimate. Since both models are evaluated on the same examples, a *paired* test, such as McNemar’s test below, is needed to conclude.

Comparing two models To compare two classifiers A and B on the same test set, McNemar’s test only looks at the examples on which they disagree: let n_{01} be the number of test examples misclassified by A but not by B , and n_{10} the number of examples misclassified by B but not by A . Under the null hypothesis that both classifiers have the same error rate, the statistic

$$\chi^2 = \frac{(|n_{01} - n_{10}| - 1)^2}{n_{01} + n_{10}}$$

approximately follows a χ^2 distribution with one degree of freedom, and the null hypothesis is rejected at the 5% level when $\chi^2 > 3.84$. Dietterich [Dietterich \(1998\)](#) showed that this test has a low type I error, and recommended it (or a 5×2 cross-validated t -test when the algorithms can be re-trained several times). On the contrary, a standard paired t -test on the scores of a k -fold cross-validation should be avoided: the training sets of the different folds overlap, so the scores are not independent and the variance is underestimated, which leads to too many false discoveries. To compare several algorithms over several datasets, Demšar [Demšar \(2006\)](#) recommends non-parametric tests based on ranks: the Wilcoxon signed-rank test for two algorithms, and the Friedman test followed by a post-hoc Nemenyi test for more than two.

Reporting A good report describes the data (origin, size, class proportions), the pre-processing, the splitting protocol, the performance measures and why they were chosen, the searched hyper-parameter ranges, the baselines, the results with their uncertainty, and the limitations of the study (possible biases of the data, conditions under which the model should not be used). Negative results, such as a sophisticated model that does not beat a simple baseline, are also informative and deserve to be reported.

11.9 A Checklist

We end this section with a checklist that summarizes the main steps of a supervised learning study.

Checklist for a supervised learning study

- ☐ The task, the unit of prediction and the target are clearly defined, and the features are available at prediction time.
- ☐ The performance measure reflects the cost of the errors, and was chosen before looking at the results.
- ☐ The data have been explored: target distribution, missing values, duplicates, outliers, temporal or grouped structure.
- ☐ The test set is split off first, with a split that respects the structure of the data (stratified, grouped or temporal), and is used only once, at the end.
- ☐ All preprocessing steps (imputation, encoding, scaling, feature selection, re-sampling) are fitted on the training data only, ideally inside a pipeline.

- ☐ Trivial and simple baselines have been evaluated.
- ☐ Hyper-parameters (including the decision threshold) are tuned by cross-validation on the training data; the final performance is estimated on the test set or by a nested cross-validation.
- ☐ In the imbalanced case, adapted measures are used, and resampling is applied to the learning folds only.
- ☐ Results are reported with their uncertainty, and models are compared with appropriate statistical tests.
- ☐ The model has been inspected (feature importance, explanations) to detect leakages and spurious correlations.
- ☐ Seeds, code, data versions and splits are saved, so that the study can be reproduced.

To remember from Section 11

- Most of the effort of a machine learning study goes into framing the problem, understanding the data and designing a sound evaluation protocol; the performance measure is chosen first, from the costs of the errors.
- Every preprocessing step (imputation, encoding, scaling, feature selection, resampling) is fitted on the training data only, ideally inside a pipeline; the split must respect the stratified, grouped or temporal structure of the data.
- The best cross-validation score of a search is optimistic; the performance of a tuned model is estimated on an untouched test set or by nested cross-validation.
- With imbalanced data, use adapted measures, resample the learning folds only, or weight the errors; costs amount to moving the decision threshold, which is a hyper-parameter.
- Results are reported with their uncertainty and compared with paired tests (McNemar, 5×2 cross-validation); interpretation tools (permutation importance, SHAP) help to detect leakages but do not reveal causes.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

Splitting the Data and Avoiding Leakage

- Q11.1.** A categorical feature has 5 modalities. How many binary features does its one-hot encoding create, and why is the encoding $1, \dots, 5$ usually a bad idea?
- Q11.2.** True or false? The features must be standardized before learning a random forest.
- Q11.3.** A merchant identifier appears in a single training transaction, which is fraudulent. What is its target encoding computed on the training set, and why is it a problem?
- Q11.4.** Why can it be useful to add a binary feature indicating that a value was missing, in addition to imputing it?

Synthesis of the section

- Q11.5.** Why is a random split inappropriate when several transactions of the same customer are in the dataset, or when the model will be used to predict the future?
- Q11.6.** In a nested cross-validation, which folds are used to choose the hyper-parameters, and which ones to estimate the performance?
- Q11.7.** Three hyper-parameters are tuned on a grid of 10 values each, by 5-fold cross-validation. How many models are learned during the search? What could you do instead?
- Q11.8.** True or false? The best of the 200 cross-validation scores obtained during a random search is an unbiased estimate of the performance of the selected model.
- Q11.9.** Why must SMOTE be applied inside the cross-validation, and not before it?
- Q11.10.** A false negative costs nine times more than a false positive. Which threshold should be applied to a well-calibrated estimate of $\mathbb{P}(Y = 1 \mid \mathbf{x})$? And if it costs four times more?
- Q11.11.** A model is learned on a training set in which the minority class has been oversampled from 1% to 50%. Are its predicted probabilities calibrated for the real data?
- Q11.12.** Two features are exact copies of each other and the model uses both. What can be said of their permutation importances?
- Q11.13.** For a given transaction, $\phi_0 = 0.10$ and the SHAP values of three features are 0.30, -0.05 and 0.15. What is the prediction of the model for this transaction?
- Q11.14.** Two models reach 85% and 86% of accuracy on the same 1000 test examples. Can we conclude that the second one is better?
- Q11.15.** Why should a standard paired t -test on the 10 scores of a 10-fold cross-validation be avoided to compare two algorithms?

Part IV

PAC-Bayesian Learning: a Theoretical Framework for Ensemble Methods

Most ensemble methods of [Part III](#) — bagging, random forests, boosting — end with a *vote*: many base classifiers, called *voters*, each give their prediction, and the ensemble returns the (weighted) majority. Such votes often work remarkably well, much better than the voters taken one by one. This last part answers two simple questions about them:

- **Why** is a vote of mediocre classifiers so good?
- **How** can we *guarantee* that a vote is good, using only the training data, without setting aside a test set?

The tool is *PAC-Bayesian theory*. It looks at an ensemble as a *probability distribution* Q over the voters (the weights of the vote), and it gives bounds of the form

$$\text{true risk} \leq \text{empirical risk} + \text{a price for having chosen } Q \text{ with the data,}$$

which hold with high probability and can be computed from the training sample.

What you need. The whole part only uses the following notions: the true and empirical risks ([Section 6](#)), expectations and Markov’s inequality ([Section 1](#)), and logarithms and exponentials. Every proof is written step by step, and the proofs that are longer than a few lines are left aside: we give their idea and a reference. The part is deliberately short; the goal is to understand the ideas and to be able to compute a bound, not to know every variant of the theory.

Plan. The part has six sections, to be read in this order ([Fig. 107](#)).

- [Section 12](#) presents the puzzling experiment that motivates the part, and the two ways of using a distribution Q over voters: the majority vote and the Gibbs classifier.
- [Section 13](#) explains how to turn an error measured on a sample into a *guarantee* on the true error, for one classifier and then for a classifier chosen among several (Occam’s bound).
- [Section 14](#) proves a first PAC-Bayesian bound, for a whole distribution Q over the voters, and shows how to read and compute it.
- [Section 15](#) goes from the Gibbs classifier to the majority vote, and shows that the quality of a vote depends on the *diversity* of its voters.
- [Section 16](#) uses the bounds as learning objectives: we learn the weights of a vote by minimizing a bound, and obtain a guarantee for free.

- **Section 17** derives such a self-bounding algorithm step by step, from the tandem bound to the learned weights of our forest.

How to read this part. As in the rest of the notes, each section opens with a box *About this section* and ends with a summary *To remember* followed by short questions *Check your understanding*, answered in **Section A**. Orange boxes *In a nutshell* give the intuition behind a result, and teal boxes *Running example* follow a single experiment, a random forest on handwritten digits, from the first to the last section. In long computations a grey arrow $\downarrow$ like this explains how each line is obtained from the previous one. The notation is summarized in **Table 7**.

Further reading. For a gentle introduction to the literature, we recommend **Alquier (2024)**, **Guedj (2019)** and **Langford (2005)**; the analysis of majority votes follows mainly **Germain et al. (2015)** and **Masegosa et al. (2020)**.

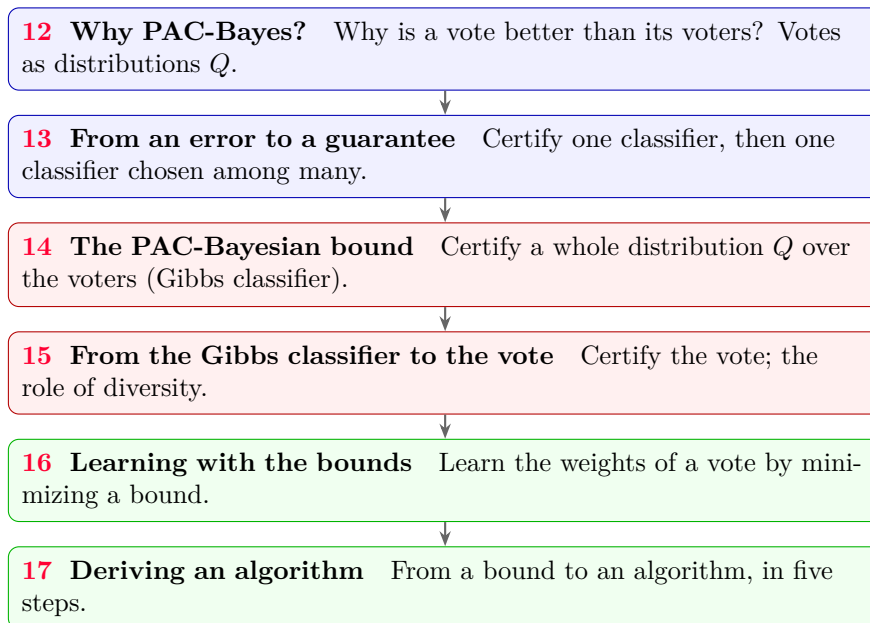


Figure 107: The six sections of **Part IV** and the question answered by each of them. `figures/tikz/roadmap.tex`

[code:

Table 7: Notation of **Part IV**.

$\mathcal{X}, \mathcal{Y} = \{-1, +1\}$	input space, output space (binary classification)
$\mathcal{D}$	unknown distribution of the data on $\mathcal{X} \times \mathcal{Y}$
$S = \{(x_i, y_i)\}_{i=1}^m$	learning sample of m examples drawn independently from $\mathcal{D}$
$\mathcal{H}$	set of <i>voters</i> (base classifiers) $h : \mathcal{X} \rightarrow \{-1, +1\}$
$R_{\mathcal{D}}(h), \hat{R}_S(h)$	true risk (probability of error on a new example) and empirical risk (fraction of errors on S) of h
P, Q	<i>prior</i> (weights chosen before seeing S) and <i>posterior</i> (weights chosen with S)
B_Q, G_Q	majority vote weighted by Q , and Gibbs classifier (a voter drawn at random from Q)
$\text{KL}(Q\ P)$	Kullback–Leibler divergence: how far Q is from P
$\text{kl}(q\ p), \overline{\text{kl}}^{-1}$	binary kl divergence and its inverse (Section 13.4)
$e_{\mathcal{D}}(Q), d_{\mathcal{D}}(Q)$	joint error and disagreement of two voters drawn from Q
δ	confidence parameter: a bound holds with probability at least $1 - \delta$ (e.g. $\delta = 0.05$)

12 Why PAC-Bayes for ensemble methods?

About this section

Guiding question. *Why is a vote of mediocre classifiers so good, and can we guarantee its accuracy without a test set?*

We start from an experiment: a random forest whose trees are mediocre, but whose vote is excellent. We then observe that bagging, random forests and boosting all produce the same object, a probability distribution Q over voters, which can be used in two ways: the majority vote and the Gibbs classifier. We finally explain what a guarantee on the error of a classifier is, and give the general shape of the PAC-Bayesian guarantees studied in this part.

12.1 A puzzling experiment

Let us start with an experiment that will accompany us throughout this part. We train a random forest of 100 trees (Section 10.2.1), each tree being learned on a bootstrap sample of half the size of the training set, on the `digits` dataset of `scikit-learn`, where the task is to decide whether a handwritten digit is smaller than 5. We use 1257 examples for training and keep 540 for testing.

Taken one by one, the trees are mediocre: their test errors range from 14% to 25%, with an average of 19%. Yet the forest, which simply lets the 100 trees vote, errs on only 4.4% of the test digits. The vote is more than four times better than an average tree, and three times better than the best one (Fig. 108).

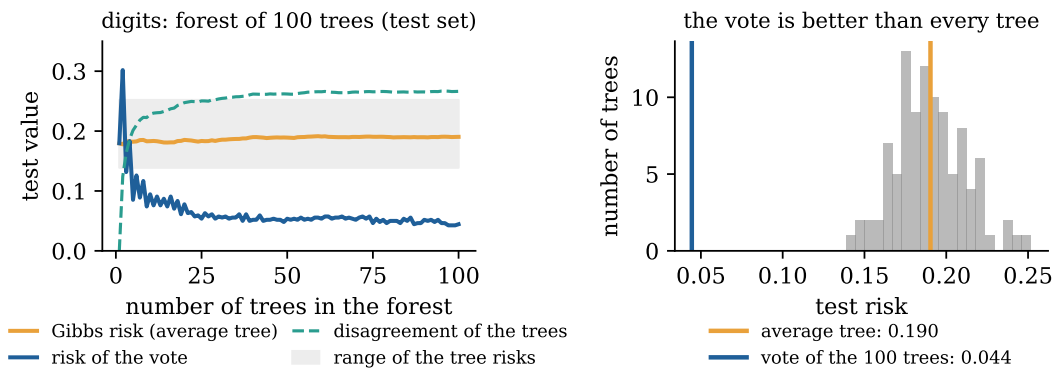


Figure 108: The running example. Left: test error of the vote of the first n trees, average test error of these trees (the *Gibbs risk*, defined below) and their disagreement (Section 15), as n grows. Right: the test errors of the 100 trees; the vote is better than every one of them. [code: `make_figures.py: fig_forest_growth`]

This experiment raises two questions, which organize the whole part.

1. **An explanatory question.** Why is a vote of mediocre classifiers so much better than its members? What property of the trees makes this possible?
2. **A practical question.** Could we have *guaranteed* that the forest is accurate *without* a test set, that is from the 1257 training examples only?

Answering the second question would be very useful: the 540 test examples could then be used for training, and we would still know how good the forest is.

Running example We will come back to this forest in every section. In [Section 13](#) we certify it with a test set, in [Section 14](#) we certify the error of an average tree without a test set, in [Section 15](#) we certify the vote itself and explain its accuracy by the diversity of the trees, and in [Section 16](#) we learn new weights for the trees by minimizing a certificate.

12.2 A vote is a distribution over voters

Ensembles produce votes. The ensemble methods of [Section 10.2](#) rely on different ideas, but they all end with the same kind of predictor. We call the base classifiers $h : \mathcal{X} \rightarrow \{-1, +1\}$ *voters*.

- In **bagging** and **random forests** ([Breiman, 1996, 2001](#)), each tree h_t is learned on a bootstrap sample, and the trees vote with the same weight: $H(x) = \text{sign}(\frac{1}{n} \sum_{t=1}^n h_t(x))$.
- In **AdaBoost**, the weak learner h_t receives a weight $\alpha_t \geq 0$ and $H(x) = \text{sign}(\sum_t \alpha_t h_t(x))$. Dividing the score by the positive number $\sum_s \alpha_s$ does not change its sign, so $H(x) = \text{sign}(\sum_t Q(h_t) h_t(x))$ with $Q(h_t) = \alpha_t / \sum_s \alpha_s$.

In both cases the weights are non-negative and sum to one: they form a *probability distribution* Q over the voters ([Fig. 109](#)). From now on, *an ensemble is a distribution* Q over a set $\mathcal{H}$ of voters, and learning an ensemble means choosing Q .

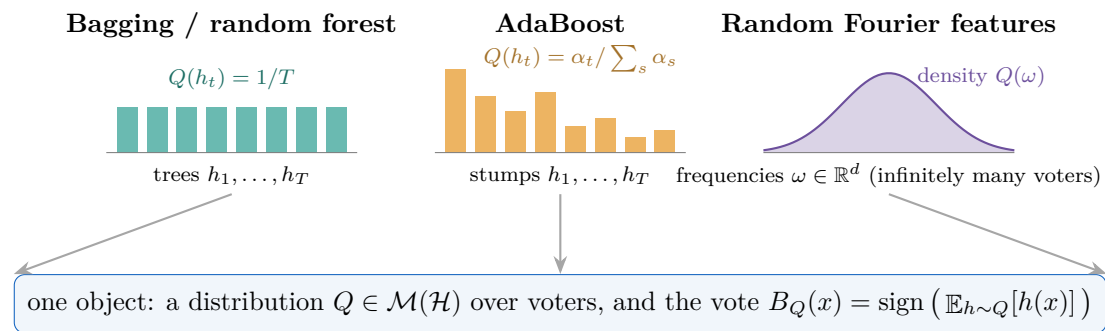


Figure 109: Three ensemble methods, one object. Bagging and random forests put the same weight on every tree, AdaBoost normalizes the weights α_t of its weak learners, and a kernel machine with random features draws its voters from a continuous distribution. In each case the predictor is the Q -weighted majority vote B_Q . [code: figures/tikz/ensembles_as_q.tex]

Two ways of using Q . A distribution Q over the voters can be used to predict in two different ways.

Definition 12.1: Majority vote and Gibbs classifier

Let Q be a probability distribution over a set $\mathcal{H}$ of voters.

- The **majority vote** B_Q predicts the label chosen by the (weighted) majority of the voters:

$$B_Q(x) = \text{sign} \left(\mathbb{E}_{h \sim Q} [h(x)] \right) = \text{sign} \left(\sum_{h \in \mathcal{H}} Q(h) h(x) \right).$$

- The **Gibbs classifier** G_Q draws *one* voter h at random according to Q , and predicts $h(x)$. A new voter is drawn for each prediction.

The Gibbs classifier is a strange way of predicting, and nobody uses it in practice. But its risk is very simple: since a voter is drawn at random for each prediction, the probability that G_Q errs is the *average* of the risks of the voters,

$$R_{\mathcal{D}}(G_Q) = \mathbb{E}_{h \sim Q} [R_{\mathcal{D}}(h)] = \sum_{h \in \mathcal{H}} Q(h) R_{\mathcal{D}}(h), \quad \hat{R}_S(G_Q) = \mathbb{E}_{h \sim Q} [\hat{R}_S(h)]. \quad (12.1)$$

In the running example, Q is uniform over the 100 trees, G_Q is “pick a tree at random and use it”, and its test error is 19%, the error of an average tree. The vote B_Q is the forest, with a test error of 4.4%.

Example 12.1 (Three voters). Consider three voters h_1, h_2, h_3 with the same weight $\frac{1}{3}$ and four equally likely points $x_1, \dots, x_4$, all of label +1. The table gives the predictions; an error is shown in red.

	x_1	x_2	x_3	x_4	risk
h_1	−1	+1	+1	+1	0.25
h_2	+1	−1	+1	+1	0.25
h_3	+1	+1	−1	+1	0.25
vote B_Q	+1	+1	+1	+1	0

Each voter errs on one point out of four, so the Gibbs risk is $R_{\mathcal{D}}(G_Q) = \frac{1}{3}(0.25 + 0.25 + 0.25) = 0.25$. But the voters never err on the *same* point: on each point, at least two voters out of three are right, and the vote never errs. If, instead, the three voters had all erred on x_1 , the vote would have erred on x_1 too, and its risk would have been 0.25, the same as the Gibbs risk.

This small example contains the answer to our explanatory question, which [Section 15](#) will make precise: *a vote is good when its voters do not make their mistakes on the same points*, that is, when they are *diverse*. The Gibbs risk only sees the average quality of the voters; the risk of the vote also depends on how their errors are spread. [Figure 110](#) shows the same phenomenon on a two-dimensional problem: individual trees are irregular and inaccurate, their vote is smooth and accurate.

In a nutshell The Gibbs risk is the average error of the voters: it is easy to study, because it is an average. The vote is what we actually use: it can be much better than the Gibbs classifier when the voters are diverse. The strategy of this

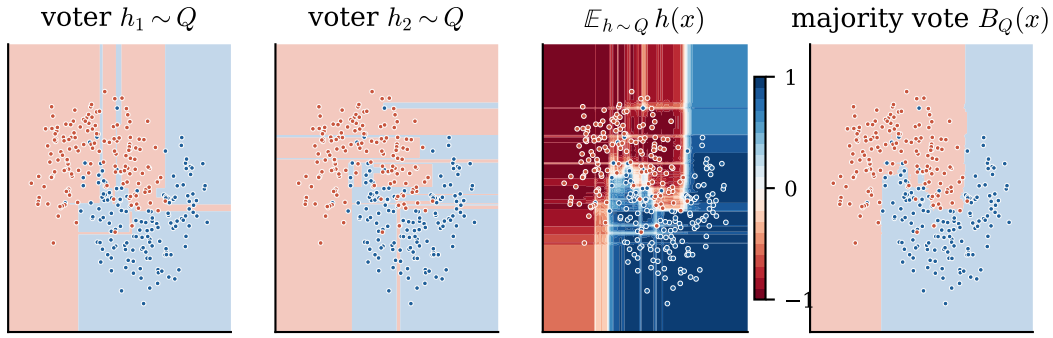


Figure 110: Two voters h_1, h_2 drawn from a random forest (uniform Q over 200 randomized trees), the average prediction $\mathbb{E}_{h \sim Q} h(x)$ and the majority vote B_Q on the `moons` dataset. Individual voters are irregular; their vote is smooth and accurate. [code: `make_figures.py: fig_gibbs_vs_mv`]

part is to first certify the Gibbs risk (Section 14), and then to go from the Gibbs classifier to the vote (Section 15).

12.3 What is a guarantee?

We now turn to the practical question. The true risk $R_{\mathcal{D}}(h)$ of a classifier is the probability that it errs on a new example. It is unknown, because the distribution $\mathcal{D}$ of the data is unknown. What we can compute is the empirical risk $\hat{R}_S(h)$, the fraction of errors on a sample S of m examples.

The sample is random. Had we been given another sample, we would have measured another empirical risk. A statement about the true risk can therefore only hold *with a certain probability over the draw of the sample*. A *guarantee*, or *risk certificate*, is a statement of the form

$$\text{w.p. at least } 1 - \delta \text{ over } S : R_{\mathcal{D}}(h) \leq \text{a number computed from } S,$$

where δ is a small number chosen by the user, typically $\delta = 0.05$. The right-hand side is called a *bound* on the true risk. “With probability at least 95%” means: if we repeated the whole experiment (draw a sample, compute the number) on many independent samples, the statement would be true for at least 95% of them. This is the PAC framework (*Probably Approximately Correct*, Valiant, 1984): *probably* refers to $1 - \delta$, *approximately* to the gap between the bound and the true risk.

Good and useless guarantees. A bound is always true with probability $1 - \delta$ (this is what a theorem proves), but it is not always useful. A bound larger than 1 is *vacuous*: a risk is always at most 1, so it says nothing. A good bound is *tight* (close to the true risk) and *computable* (every term can be computed from the data).

Why this is difficult for ensembles. If the classifier is chosen *before* looking at the sample, its empirical risk is an unbiased estimate of its true risk, and Section 13

shows how to turn it into a guarantee. The difficulty comes from learning: an ensemble is *chosen with the sample*, so it fits the sample, and its empirical risk is optimistic (Section 6). The classical answer is to pay a price for the size or the complexity of the set of classifiers we choose from (Sections 6 and 10). For ensembles this price is far too high: a forest of 100 trees is an element of a gigantic set, and the corresponding bounds are larger than 1. Moreover, they pay the same price whatever the weights we choose, and cannot see the diversity of the voters.

12.4 The PAC-Bayesian idea

PAC-Bayesian theory replaces the size of the set of classifiers by a quantity that depends on what has been learned (Fig. 111).

1. **Before** seeing the data, we fix a distribution P over the voters, called the **prior**; for instance the uniform distribution over the trees.
2. A learning algorithm looks at the sample and chooses a distribution Q , called the **posterior**; for instance the weights of the vote.
3. A bound compares Q to P : the closer Q is to P , the smaller the price.

The bounds of Section 14 have the following shape: with probability at least $1 - \delta$ over S , for all posteriors Q simultaneously,

$$\underbrace{R_{\mathcal{D}}(G_Q)}_{\text{true risk (unknown)}} \leq \underbrace{\hat{R}_S(G_Q)}_{\text{empirical risk}} + \underbrace{\text{price}(\text{KL}(Q\|P), m, \delta)}_{\text{price for having chosen } Q \text{ with } S}, \quad (12.2)$$

where $\text{KL}(Q\|P)$ measures how far the posterior is from the prior, and the price decreases when the sample size m grows.

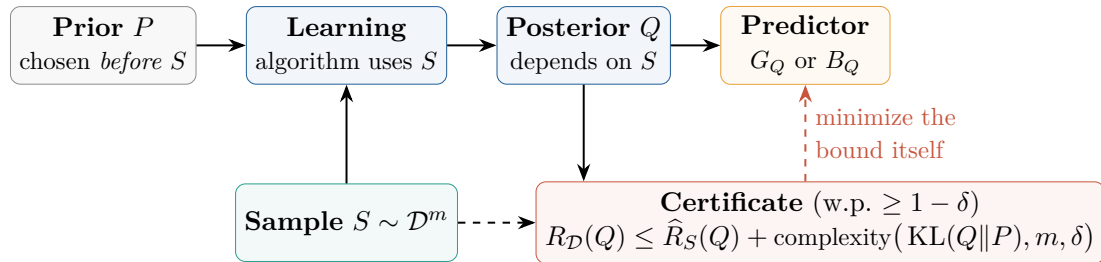


Figure 111: The PAC-Bayesian setting. The prior P is fixed before seeing the data; the posterior Q may depend on S in any way. The bound holds simultaneously for all posteriors, so it can be used both as a certificate and as an objective to minimize (Section 16). [code:

figures/tikz/principle.tex]

In a nutshell PAC-Bayes = PAC + Bayes. From the PAC framework, it inherits the form of the guarantee: a bound that holds with probability $1 - \delta$ over the sample, whatever the distribution of the data. From Bayesian statistics, it borrows the words *prior* and *posterior*. But the posterior does not need to be computed with Bayes' rule: it is *any* distribution we like, for instance the weights of a random forest or of AdaBoost.

A short history. The first PAC-Bayesian bounds are due to [Shawe-Taylor and Williamson \(1997\)](#) and [McAllester \(1998, 1999\)](#). They were made tighter by [Seeger \(2002\)](#), [Maurer \(2004\)](#) and [Catoni \(2007\)](#), and applied to majority votes by [Lacasse et al. \(2006\)](#) and [Germain et al. \(2015\)](#). They are today among the very few bounds that give non-vacuous guarantees for complex models such as random forests ([Lorenzen et al., 2019](#); [Masegosa et al., 2020](#)) and deep neural networks ([Dziugaite and Roy, 2017](#); [Pérez-Ortiz et al., 2021](#)).

To remember from Section 12

- Bagging, random forests and boosting output a *distribution* Q over voters: the weights of a majority vote.
- The *majority vote* B_Q is what we use; the *Gibbs classifier* G_Q draws one voter at random. Its risk is the average risk of the voters.
- A vote can be much better than its average voter, when the voters do not err on the same points (diversity).
- A guarantee is a bound on the true risk that holds with probability at least $1 - \delta$ over the sample, and that can be computed from the data.
- PAC-Bayesian bounds read: true risk $\leq$ empirical risk + a price that grows with the distance $\text{KL}(Q\|P)$ between the posterior Q and a prior P fixed before seeing the data.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

- Q12.1.** Three voters have risks 0.1, 0.2 and 0.3, and weights 0.5, 0.3 and 0.2. What is the risk of the Gibbs classifier?
- Q12.2.** AdaBoost outputs the weights $\alpha_1 = 1$, $\alpha_2 = 3$. What is the corresponding distribution Q ?
- Q12.3.** Can the vote be worse than the Gibbs classifier? Can it be better?
- Q12.4.** A bound states that, with probability at least 0.95, $R_{\mathcal{D}}(h) \leq 1.3$. Is it false? Is it useful?
- Q12.5.** Why must the prior be chosen before seeing the data?

13 From an error on a sample to a guarantee

About this section

Guiding question. *How can an error measured on a sample be turned into a guarantee on the true error, first for one classifier, then for a classifier chosen among many?*

We start with a single classifier fixed in advance. Its empirical risk is an average of independent random variables, and two inequalities, Markov's and Hoeffding's, tell us how far it can be from the true risk. This gives the test-set bound, and a sharper version based on the binary kl divergence. We then see why these bounds fail when the classifier is chosen with the data, and how to repair them: the union bound and, more cleverly, Occam's bound, which already contains the main idea of PAC-Bayes.

13.1 One classifier fixed in advance

Let h be a classifier chosen *before* looking at a sample $S = \{(x_i, y_i)\}_{i=1}^m$ of m examples drawn independently from $\mathcal{D}$; think of a model trained on other data and evaluated on a test set. For each example, let $Z_i = \mathbb{1}[h(x_i) \neq y_i]$ be 1 if h errs on it and 0 otherwise. Then:

- $Z_1, \dots, Z_m$ are independent, because the examples are, and h does not depend on them;
- each Z_i is a Bernoulli variable whose mean is the true risk: $\mathbb{E}Z_i = \mathbb{P}(h(x_i) \neq y_i) = R_{\mathcal{D}}(h)$;
- the empirical risk is their average, $\hat{R}_S(h) = \frac{1}{m} \sum_{i=1}^m Z_i$.

Hence $\mathbb{E} \hat{R}_S(h) = R_{\mathcal{D}}(h)$: the empirical risk is an unbiased estimate of the true risk, and its variance is $R_{\mathcal{D}}(h)(1 - R_{\mathcal{D}}(h))/m$, which goes to 0 when m grows. The question is: *how large can the gap $R_{\mathcal{D}}(h) - \hat{R}_S(h)$ be, with a given probability?* Measuring $\hat{R}_S(h)$ is like flipping m times a coin whose bias $R_{\mathcal{D}}(h)$ is unknown, and asking how far the observed frequency can be from the bias.

13.2 Markov's inequality

All the bounds of this part rely on one very simple inequality, already met in [Section 1](#) (Proposition [1.9](#)).

Lemma 13.1: Markov's inequality

Let $Z \geq 0$ be a non-negative random variable and $a > 0$. Then

$$\mathbb{P}(Z \geq a) \leq \frac{\mathbb{E}Z}{a}.$$

Equivalently, for any $\delta \in (0, 1]$, with probability at least $1 - \delta$, $Z \leq \frac{\mathbb{E}Z}{\delta}$.

Proof. Since $Z \geq 0$, we have $a\mathbb{1}[Z \geq a] \leq Z$ for every value of Z ([Fig. 112](#)): the left-hand side is 0 when $Z < a$, and it is $a \leq Z$ otherwise. Taking expectations,

$$a\mathbb{P}(Z \geq a) = \mathbb{E}[a\mathbb{1}[Z \geq a]]$$

↓ expectation of an indicator: $\mathbb{E}\mathbf{1}[A] = \mathbb{P}(A)$; then $a\mathbf{1}[Z \geq a] \leq Z$
 $\leq \mathbb{E}Z$.

For the second form, apply the first one with $a = \mathbb{E}Z/\delta$: then $\mathbb{P}(Z \geq \mathbb{E}Z/\delta) \leq \delta$. $\square$

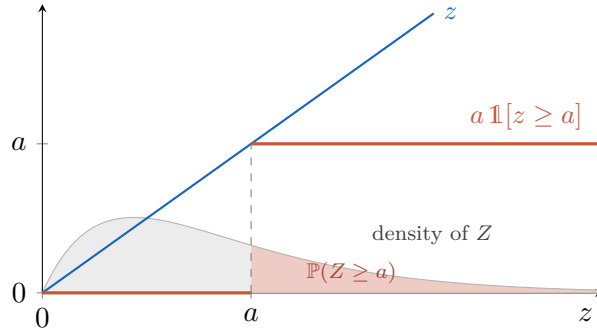


Figure 112: Why Markov's inequality holds: for a non-negative variable, the step $z \mapsto a\mathbf{1}[z \geq a]$ (red) lies below the identity $z \mapsto z$ (blue), so that $a\mathbb{P}(Z \geq a) \leq \mathbb{E}Z$. The shaded red area is the tail probability. [code: figures/tikz/markov.tex]

Example 13.1. If the average salary in a company is 2000 euros, at most 10% of the employees earn 20 000 euros or more: otherwise these employees alone would bring the average above $0.1 \times 20\,000 = 2000$ euros. This is Markov's inequality with $a = 10\mathbb{E}Z$.

Markov's inequality is weak when applied directly, but it becomes very powerful when applied to a well-chosen function of the variable of interest, such as an exponential. This is how all the bounds below are obtained.

13.3 Hoeffding's inequality and the test-set bound

Theorem 13.2: Hoeffding's inequality (Hoeffding, 1963)

Let $Z_1, \dots, Z_m$ be independent random variables with values in $[0, 1]$ and the same mean p , and let $\hat{p} = \frac{1}{m} \sum_{i=1}^m Z_i$. For every $\varepsilon > 0$,

$$\mathbb{P}(p - \hat{p} \geq \varepsilon) \leq e^{-2m\varepsilon^2}.$$

The proof is given in [Section 1](#) (Proposition 1.14). Its idea is worth remembering, because the PAC-Bayesian proof of [Section 14](#) uses the same ingredients. For any $\lambda > 0$, the event $p - \hat{p} \geq \varepsilon$ is the same as $e^{\lambda(p-\hat{p})} \geq e^{\lambda\varepsilon}$, so that Markov's inequality gives $\mathbb{P}(p - \hat{p} \geq \varepsilon) \leq e^{-\lambda\varepsilon} \mathbb{E} e^{\lambda(p-\hat{p})}$. The expectation of the exponential is then bounded thanks to independence and to *Hoeffding's lemma* (Lemma 1.13):

$$\mathbb{E} e^{\lambda(p-\hat{p})} \leq e^{\lambda^2/(8m)} \quad \text{for every } \lambda \in \mathbb{R}, \quad (13.1)$$

and choosing $\lambda = 4m\varepsilon$ gives the result.

Hoeffding's inequality says that the empirical risk underestimates the true risk by more than ε with probability at most $e^{-2m\varepsilon^2}$. Turned around, it gives a guarantee.

Corollary 13.3: Test-set bound

Let h be a classifier chosen independently of the sample $S \sim \mathcal{D}^m$, and $\delta \in (0, 1]$. With probability at least $1 - \delta$ over S ,

$$R_{\mathcal{D}}(h) \leq \hat{R}_S(h) + \sqrt{\frac{\ln(1/\delta)}{2m}}.$$

Proof. Apply Theorem 13.2 to $Z_i = \mathbb{1}[h(x_i) \neq y_i]$, with $p = R_{\mathcal{D}}(h)$, $\hat{p} = \hat{R}_S(h)$, and choose ε such that the right-hand side equals δ :

$$e^{-2m\varepsilon^2} = \delta \iff 2m\varepsilon^2 = \ln \frac{1}{\delta} \iff \varepsilon = \sqrt{\frac{\ln(1/\delta)}{2m}}.$$

Then $\mathbb{P}(R_{\mathcal{D}}(h) - \hat{R}_S(h) \geq \varepsilon) \leq \delta$: with probability at least $1 - \delta$, $R_{\mathcal{D}}(h) < \hat{R}_S(h) + \varepsilon$. $\square$

Example 13.2 (Reading the test-set bound). A classifier makes 100 errors on a test set of $m = 1000$ examples: $\hat{R}_S(h) = 0.1$. With $\delta = 0.05$, $\ln(1/\delta) = \ln 20 \approx 3.0$ and

$$R_{\mathcal{D}}(h) \leq 0.1 + \sqrt{\frac{3.0}{2000}} \approx 0.1 + 0.039 = 0.139.$$

With 95% confidence, the true risk is below 13.9%. With 4000 test examples, the margin 0.039 would be halved: it decreases like $1/\sqrt{m}$. A higher confidence costs little: $\delta = 0.001$ gives $\ln(1/\delta) \approx 6.9$ and a margin of 0.059.

Running example Suppose that we had fixed the forest before seeing the 540 test digits, and that we use them only to certify it. The vote errs on 4.4% of them; with $\delta = 0.05$, the test-set bound certifies a true risk below $0.044 + \sqrt{\ln 20/1080} = 0.097$. This number will serve as a reference: our goal is to obtain comparable guarantees *without* setting aside a test set.

13.4 A sharper bound with the binary kl

The margin $\sqrt{\ln(1/\delta)/(2m)}$ of the test-set bound does not depend on the risk itself. This is wasteful when the risk is small: a coin of bias 0.01 fluctuates much less than a coin of bias 0.5 (its variance $p(1-p)$ is 25 times smaller). The sharpest bounds for coins use the following quantity.

Definition 13.4: Binary kl divergence

For $q, p \in [0, 1]$, the **binary kl divergence** is

$$\text{kl}(q||p) = q \ln \frac{q}{p} + (1-q) \ln \frac{1-q}{1-p},$$

with the convention $0 \ln 0 = 0$. It is non-negative and equal to 0 only if $q = p$.

The binary kl measures how *surprising* it is to observe a frequency q of errors when the true error rate is p : the probability that m flips of a coin of bias p give a frequency

close to q behaves like $e^{-m \text{kl}(q\|p)}$. For instance $\text{kl}(0.1\|0.2) \approx 0.037$: observing at most 10% of errors on $m = 100$ examples when the true risk is 20% has a probability at most $e^{-100 \times 0.037} \approx 2.5\%$ (the exact value is 0.6%). The following theorem makes this precise (Langford, 2005); it is the kl form of Chernoff's bound, proved in Section 1 (Example 1.6).

Theorem 13.5: kl test-set bound

Let h be a classifier chosen independently of $S \sim \mathcal{D}^m$, and $\delta \in (0, 1]$. With probability at least $1 - \delta$ over S ,

$$\widehat{R}_S(h) \geq R_{\mathcal{D}}(h) \quad \text{or} \quad \text{kl}(\widehat{R}_S(h) \| R_{\mathcal{D}}(h)) \leq \frac{\ln(1/\delta)}{m}.$$

Turning a kl into a bound. The theorem bounds a kl, not the risk. To get a bound on the risk, we look for the largest p that is still compatible with the observation. For a fixed q , the function $p \mapsto \text{kl}(q\|p)$ is 0 at $p = q$ and increases on $[q, 1]$ (Fig. 113), so the set of $p \geq q$ such that $\text{kl}(q\|p) \leq \varepsilon$ is an interval $[q, \overline{\text{kl}}^{-1}(q, \varepsilon)]$, where

$$\overline{\text{kl}}^{-1}(q, \varepsilon) = \max \{p \in [q, 1] : \text{kl}(q\|p) \leq \varepsilon\}.$$

Theorem 13.5 then reads: with probability at least $1 - \delta$,

$$R_{\mathcal{D}}(h) \leq \overline{\text{kl}}^{-1}\left(\widehat{R}_S(h), \frac{\ln(1/\delta)}{m}\right). \quad (13.2)$$

There is no formula for $\overline{\text{kl}}^{-1}$, but it is easy to compute by *bisection*, since we look for the point where an increasing function crosses a level (Algorithm 12).

Algorithm 12: Computing $\overline{\text{kl}}^{-1}(q, \varepsilon)$ by bisection

Input: empirical value $q \in [0, 1]$, level $\varepsilon \geq 0$, tolerance τ (e.g. 10^{-6})

```

1  $a \leftarrow q, b \leftarrow 1$ ; // the answer is in  $[a, b]$ 
2 while  $b - a > \tau$  do
3    $c \leftarrow (a + b)/2$ ;
4   if  $\text{kl}(q\|c) > \varepsilon$  then  $b \leftarrow c$ ;
5   else  $a \leftarrow c$ ;
```

Output: b

Example 13.3 (Inverting the kl by hand). Let $q = 0.1$ and $\varepsilon = 0.05$. Since $\text{kl}(0.1\|0.2) \approx 0.037 \leq 0.05$ and $\text{kl}(0.1\|0.25) \approx 0.072 > 0.05$, the answer lies in $[0.2, 0.25]$. At the midpoint, $\text{kl}(0.1\|0.225) \approx 0.053 > 0.05$, so the answer is in $[0.2, 0.225]$, and so on. After twenty steps, we find $\overline{\text{kl}}^{-1}(0.1, 0.05) \approx 0.220$.

The kl bound is never worse. Pinsker's inequality states that $\text{kl}(q\|p) \geq 2(q - p)^2$ for all q, p . Hence, if $\text{kl}(q\|p) \leq \varepsilon$, then $2(p - q)^2 \leq \varepsilon$, that is $p \leq q + \sqrt{\varepsilon/2}$:

$$\overline{\text{kl}}^{-1}(q, \varepsilon) \leq q + \sqrt{\varepsilon/2}.$$

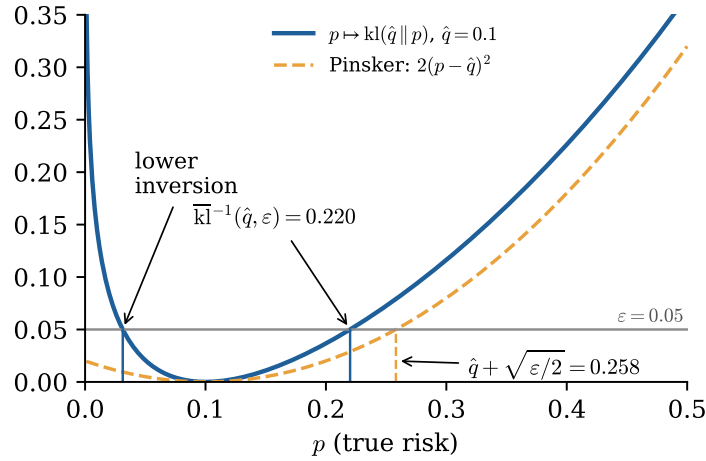


Figure 113: The function $p \mapsto \text{kl}(\hat{q} \| p)$ for $\hat{q} = 0.1$. For $\varepsilon = 0.05$, the values of p compatible with $\text{kl}(\hat{q} \| p) \leq \varepsilon$ form an interval whose upper end $\bar{\text{kl}}^{-1}(\hat{q}, \varepsilon) = 0.220$ is smaller than the Hoeffding-type value $\hat{q} + \sqrt{\varepsilon/2} = 0.258$.
[code: make_figures.py: fig_kl_inverse]

With $\varepsilon = \ln(1/\delta)/m$, the right-hand side is exactly the test-set bound of Corollary 13.3. The kl bound is therefore always at least as good, and much better when the empirical risk is small (Fig. 114): for $\hat{R}_S(h) = 0.01$, $m = 1000$ and $\delta = 0.05$, Hoeffding gives 0.049 and the kl 0.020.

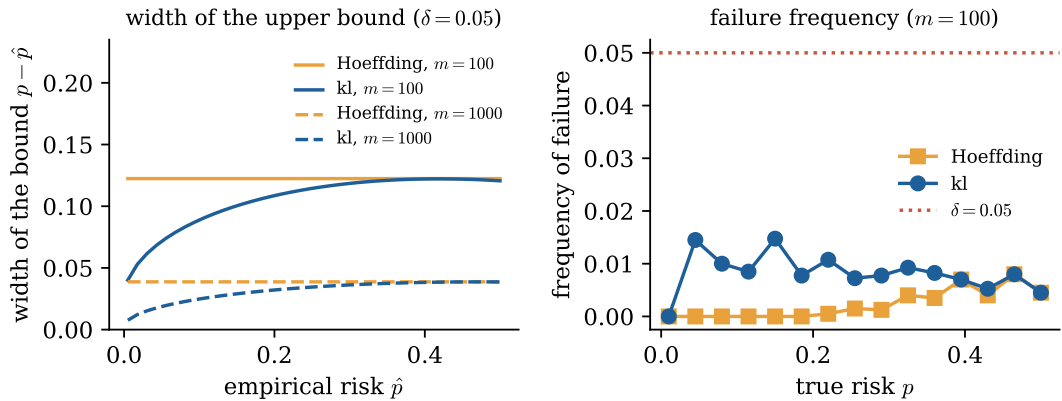


Figure 114: Left: margin $p - \hat{p}$ of the upper bound on the risk of a fixed classifier given by Hoeffding's inequality (constant) and by the kl, as a function of the empirical risk, for $m = 100$ and $m = 1000$. Right: frequency, over 4000 simulated samples of size 100, of the event “the true risk is above the bound”; it stays below $\delta = 0.05$ for both bounds: they are valid.
[code: make_figures.py: fig_concentration]

Running example With the kl, the 540 test digits certify the vote of the forest at $\bar{\text{kl}}^{-1}(0.044, \ln 20/540) = 0.069$ instead of 0.097. For an average tree (19% of test errors), the two certificates are 0.234 (kl) and 0.243 (Hoeffding): the kl makes a difference for small risks only.

13.5 What goes wrong when we choose

The bounds above require the classifier to be fixed *before* seeing the sample. Let us see what happens if this is not the case. Take $K = 1000$ classifiers that answer at random (true risk 0.5 each), evaluate them on the same sample of $m = 100$ examples, and keep the one with the smallest empirical risk. Among 1000 random classifiers, some are lucky on this sample: the selected one typically shows an empirical risk of about 0.34 (Fig. 115). The test-set bound, applied naively to it, gives $0.34 + 0.12 = 0.46 < 0.5$: it is *wrong* in almost all the runs of the experiment.

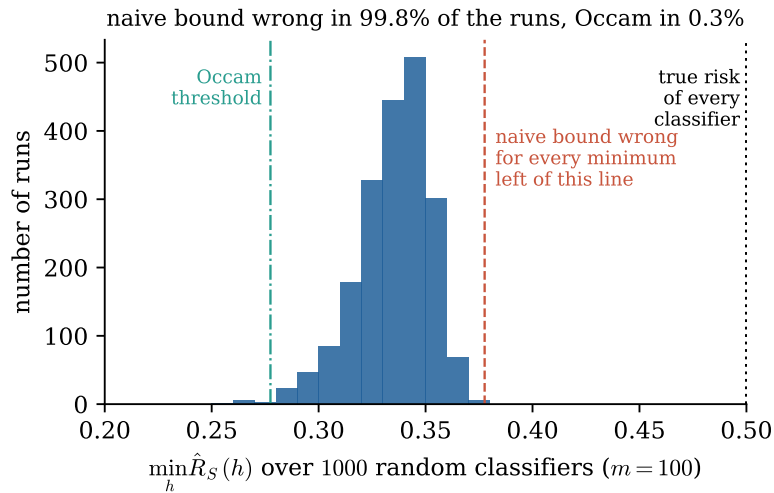


Figure 115: Selecting the best of 1000 random classifiers (true risk 0.5) on $m = 100$ examples, over 1000 repetitions. Histogram: empirical risk of the selected classifier. Vertical lines: the largest empirical risk for which the naive test-set bound, and the bound with the union bound (Occam's bound with a uniform prior), stay below the true risk 0.5. The naive bound is wrong in almost every run, the union bound almost never. [code: make_figures.py: fig_uniformity]

Pitfall The test-set bound holds for a classifier fixed in advance, *not* for the classifier that looks best on the sample. Choosing is learning, and learning makes the empirical risk optimistic. This is exactly the situation of the training error of a learned model, and of the best score of a cross-validation (Section 7).

The repair is to ask for a guarantee that holds *for all classifiers at once*: then it holds in particular for the chosen one, whatever the way it was chosen. The tool is the following.

Lemma 13.6: Union bound

For any events $A_1, \dots, A_K$, $\mathbb{P}(A_1 \cup \dots \cup A_K) \leq \sum_{k=1}^K \mathbb{P}(A_k)$.

In words: the probability that *at least one* of several bad events happens is at most the sum of their probabilities.

Theorem 13.7: Finite class bound

Let $\mathcal{H} = \{h_1, \dots, h_K\}$ be a finite set of classifiers fixed before seeing $S \sim \mathcal{D}^m$. With probability at least $1 - \delta$, for all $h \in \mathcal{H}$,

$$R_{\mathcal{D}}(h) \leq \hat{R}_S(h) + \sqrt{\frac{\ln K + \ln(1/\delta)}{2m}}.$$

Proof. For each k , let A_k be the bad event “the test-set bound fails for h_k at level δ/K ”. Each h_k is fixed in advance, so Corollary 13.3 applied with δ/K gives $\mathbb{P}(A_k) \leq \delta/K$. By the union bound,

$$\mathbb{P}(\text{at least one } A_k) \leq \sum_{k=1}^K \frac{\delta}{K} = \delta.$$

With probability at least $1 - \delta$, no bad event happens, that is, for all k , $R_{\mathcal{D}}(h_k) \leq \hat{R}_S(h_k) + \sqrt{\ln(K/\delta)/(2m)}$, and $\ln(K/\delta) = \ln K + \ln(1/\delta)$. $\square$

The price of choosing among K classifiers is only $\ln K$ in the numerator: with $m = 1000$, an empirical risk of 0.1 and $\delta = 0.05$, the bound is 0.139 for one classifier fixed in advance, 0.170 for the best of $K = 1000$, and 0.192 for the best of a million.

13.6 Occam’s bound: spreading the confidence with a prior

In the proof above, every classifier receives the same share δ/K of the confidence budget δ . Nothing forces us to share it equally: we can give a larger share to the classifiers we believe in, *as long as this belief is fixed before seeing the data*. This belief is described by a *prior*: a probability distribution P over $\mathcal{H}$, with $P(h) \geq 0$ and $\sum_h P(h) = 1$.

Theorem 13.8: Occam’s bound (Blumer et al., 1987; McAllester, 1998)

Let $\mathcal{H}$ be a finite (or countable) set of classifiers and P a prior on $\mathcal{H}$, both fixed before seeing $S \sim \mathcal{D}^m$. With probability at least $1 - \delta$, for all $h \in \mathcal{H}$ with $P(h) > 0$,

$$R_{\mathcal{D}}(h) \leq \hat{R}_S(h) + \sqrt{\frac{\ln \frac{1}{P(h)} + \ln \frac{1}{\delta}}{2m}}.$$

Proof. Same proof as Theorem 13.7, with the share $\delta P(h)$ instead of δ/K for each h . The test-set bound at level $\delta P(h)$ fails for h with probability at most $\delta P(h)$, so the probability that it fails for at least one h is at most $\sum_h \delta P(h) = \delta$. Finally $\ln \frac{1}{\delta P(h)} = \ln \frac{1}{P(h)} + \ln \frac{1}{\delta}$. $\square$

With the uniform prior $P(h) = 1/K$, $\ln \frac{1}{P(h)} = \ln K$ and we recover Theorem 13.7. With a non-uniform prior, the classifiers with a large prior weight pay less, and those with a small prior weight pay more.

Example 13.4 (A favourite classifier). We choose among $K = 1000$ classifiers, $m = 1000$, $\delta = 0.05$. Before seeing the data, an expert tells us which classifier she expects to be the best. We give it the prior weight $\frac{1}{2}$ and share the other half among the 999 others

($P(h) = \frac{1}{1998}$). If the favourite classifier has an empirical risk of 0.08, its certificate is $0.08 + \sqrt{(\ln 2 + \ln 20)/2000} = 0.123$, better than with the uniform prior (0.150). If another classifier is selected with the same empirical risk, its certificate is 0.153: a little worse than with the uniform prior. A good prior helps a lot; a bad one costs little.

In a nutshell The prior is not a belief about the world, it is a way of *spreading the confidence budget* δ over the classifiers. Any prior gives a valid bound, as long as it is fixed before seeing the data; a good prior gives a tight one. The price $\ln \frac{1}{P(h)}$ is the number of nats needed to describe h with a code built from P : simple classifiers, which get a large prior weight, are cheap. This is a mathematical form of *Occam's razor*: among the explanations of the data, prefer the simplest.

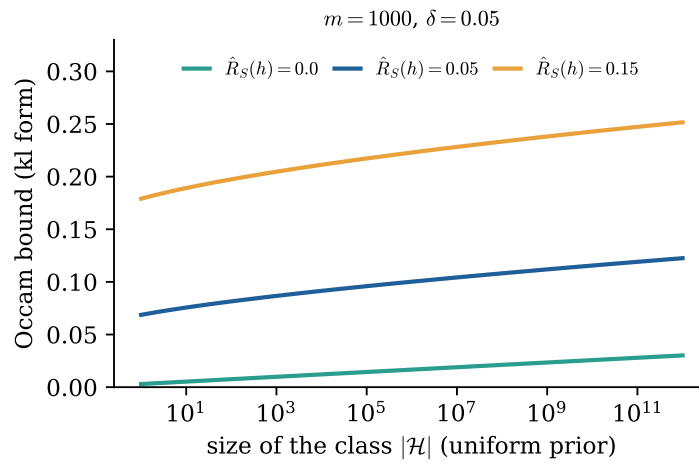


Figure 116: Occam's bound (kl form) as a function of the size $|\mathcal{H}|$ of the class, with the uniform prior, for $m = 1000$, $\delta = 0.05$ and three empirical risks. The price grows only like $\ln |\mathcal{H}|$: certifying a classifier chosen among 10^{10} is still possible. [code: make_figures.py: fig_occam]

Towards PAC-Bayes. Occam's bound certifies *one* classifier, chosen with the data, and the price is $\ln \frac{1}{P(h)}$. A vote uses a whole *distribution* Q over the classifiers. The next section shows that the price $\ln \frac{1}{P(h)}$ is a special case of a quantity defined for any distribution Q , the Kullback–Leibler divergence $\text{KL}(Q\|P)$, and that Occam's bound is a special case of the PAC-Bayesian bound.

To remember from Section 13

- For a classifier fixed in advance, the empirical risk is an average of m independent Bernoulli variables, of mean the true risk.
- *Markov's inequality* $\mathbb{P}(Z \geq a) \leq \mathbb{E}Z/a$ is the basic tool; applied to an exponential it gives *Hoeffding's inequality*.
- *Test-set bound*: $R_{\mathcal{D}}(h) \leq \hat{R}_S(h) + \sqrt{\ln(1/\delta)/(2m)}$ with probability $1 - \delta$. The *kl bound* $R_{\mathcal{D}}(h) \leq \bar{\text{kl}}^{-1}(\hat{R}_S(h), \ln(1/\delta)/m)$ is never worse, and much better for small risks; $\bar{\text{kl}}^{-1}$ is computed by bisection.
- These bounds fail for a classifier *chosen* with the data. The *union bound*

repairs them at the price $\ln K$ for K classifiers.

- *Occam's bound* spreads the confidence with a prior P fixed in advance: the price becomes $\ln \frac{1}{P(h)}$.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

- Q13.1.** A non-negative random variable has mean 0.02. What can you say about $\mathbb{P}(Z \geq 0.2)$?
- Q13.2.** A classifier fixed in advance makes 30 errors on $m = 600$ test examples. Give its test-set bound at level $\delta = 0.05$.
- Q13.3.** How many test examples are needed for a margin of 0.01 with $\delta = 0.05$ in the test-set bound?
- Q13.4.** Why is the kl bound much better than the test-set bound when the empirical risk is close to 0?
- Q13.5.** Why can't we apply the test-set bound to the classifier with the smallest training error?
- Q13.6.** With the uniform prior on K classifiers, what is the price $\ln \frac{1}{P(h)}$ in Occam's bound? What happens if K is multiplied by 10?

14 The PAC-Bayesian bound

About this section

Guiding question. *How can we certify a whole distribution Q over voters, chosen with the data?*

We go from Occam's bound, which certifies one classifier, to a bound on the risk of the Gibbs classifier G_Q , for any distribution Q . We first define the Kullback–Leibler divergence $\text{KL}(Q\|P)$, which measures how far Q is from the prior. We then prove, in three steps, a first PAC-Bayesian bound, and state the two bounds used in practice, McAllester's and Seeger's. We end by reading the bound, and by finding the posterior that minimizes it.

Throughout the section, $\mathcal{H}$ is a *finite* set of n voters, so that a distribution Q over $\mathcal{H}$ is just a list of n non-negative weights $Q(h_1), \dots, Q(h_n)$ summing to one. Everything extends to infinite sets of voters (sums become integrals), but we will not need it. Recall from [Section 12](#) that the risks of the Gibbs classifier are averages of the risks of the voters:

$$R_{\mathcal{D}}(G_Q) = \sum_{h \in \mathcal{H}} Q(h) R_{\mathcal{D}}(h), \quad \hat{R}_S(G_Q) = \sum_{h \in \mathcal{H}} Q(h) \hat{R}_S(h).$$

14.1 Measuring the distance to the prior: the KL divergence

In Occam's bound, choosing the classifier h costs $\ln \frac{1}{P(h)}$. We now look for the price of choosing a whole distribution Q .

Definition 14.1: Kullback–Leibler divergence

Let P and Q be two distributions on a finite set $\mathcal{H}$, with $P(h) > 0$ for all h . The **Kullback–Leibler (KL) divergence** of Q from P is

$$\text{KL}(Q\|P) = \sum_{h \in \mathcal{H}} Q(h) \ln \frac{Q(h)}{P(h)} = \mathbb{E}_{h \sim Q} \left[\ln \frac{Q(h)}{P(h)} \right],$$

with the convention $0 \ln 0 = 0$ (voters with $Q(h) = 0$ do not count).

The KL divergence is the average, over the voters chosen by Q , of the log-ratio $\ln \frac{Q(h)}{P(h)}$: how much more weight the posterior gives to a voter than the prior did. It is not a distance in the mathematical sense (it is not symmetric), but it behaves like one.

Proposition 14.2: Properties of the KL divergence

For all distributions P, Q on $\mathcal{H}$: $\text{KL}(Q\|P) \geq 0$, and $\text{KL}(Q\|P) = 0$ if and only if $Q = P$.

Proof. Let $\mathcal{H}_Q = \{h : Q(h) > 0\}$. The logarithm is concave, so Jensen's inequality ([Proposition 1.3](#)) gives $\mathbb{E}[\ln U] \leq \ln \mathbb{E}[U]$ for any positive random variable U . With

$U = \frac{P(h)}{Q(h)}$ and $h \sim Q$,

$$\begin{aligned}
 -\text{KL}(Q\|P) &= \sum_{h \in \mathcal{H}_Q} Q(h) \ln \frac{P(h)}{Q(h)} \\
 &\stackrel{\downarrow \text{ Jensen's inequality }}{\leq} \ln \sum_{h \in \mathcal{H}_Q} Q(h) \frac{P(h)}{Q(h)} = \ln \sum_{h \in \mathcal{H}_Q} P(h) \\
 &\stackrel{\downarrow \text{ the weights of } P \text{ sum to at most one on } \mathcal{H}_Q}{\leq} \ln 1 = 0.
 \end{aligned}$$

Equality holds when $\frac{P(h)}{Q(h)}$ is constant on $\mathcal{H}_Q$ and $\mathcal{H}_Q = \mathcal{H}$, that is when $Q = P$. $\square$

Example 14.1 (Three typical posteriors). Let P be uniform on n voters: $P(h) = \frac{1}{n}$.

- $Q = P$ (we keep the prior): $\text{KL}(Q\|P) = 0$.
- Q **uniform on k of the n voters** (for instance the k best ones): each of the k voters has $Q(h) = \frac{1}{k}$, so $\text{KL}(Q\|P) = \sum_{k \text{ voters}} \frac{1}{k} \ln \frac{1/k}{1/n} = \ln \frac{n}{k}$.
- Q **puts all its weight on one voter h** (a *Dirac* distribution δ_h): $\text{KL}(\delta_h\|P) = 1 \times \ln \frac{1}{1/n} = \ln n$. More generally, $\text{KL}(\delta_h\|P) = \ln \frac{1}{P(h)}$: this is exactly the price of Occam's bound.

With $n = 100$ voters: keeping the uniform distribution costs 0, spreading the weight over the 10 best voters costs $\ln 10 \approx 2.3$, and choosing the single best voter costs $\ln 100 \approx 4.6$. *The more concentrated the posterior, the higher the price.*

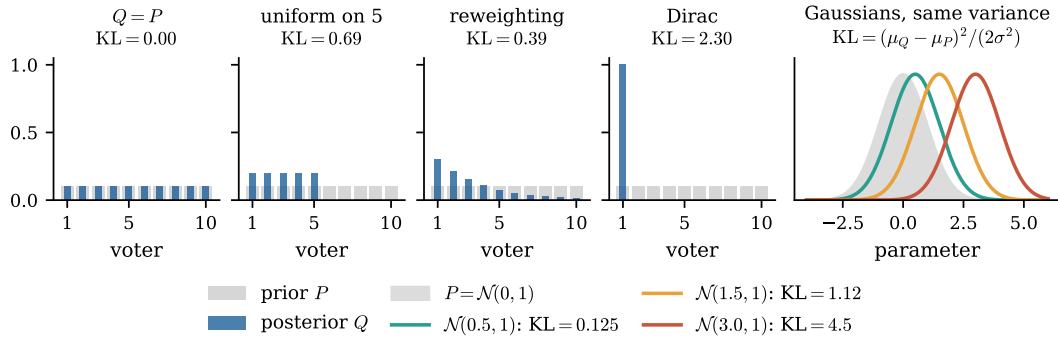


Figure 117: Examples of priors P (grey) and posteriors Q (blue) on 10 voters, and the corresponding KL divergence. The KL is 0 when $Q = P$ and grows as Q concentrates on a few voters that P considered unlikely. Right: the same for Gaussian distributions on a continuous parameter. [code: make_figures.py: fig_kl_examples]

14.2 The change of measure

The key step of every PAC-Bayesian proof is an inequality that allows to replace an average over the posterior Q , which depends on the data, by an average over the prior P , which does not.

Lemma 14.3: Change of measure (Donsker and Varadhan, 1975)

Let P, Q be distributions on a finite set $\mathcal{H}$ with $P(h) > 0$ for all h . For every function $\phi : \mathcal{H} \rightarrow \mathbb{R}$,

$$\mathbb{E}_{h \sim Q} [\phi(h)] \leq \text{KL}(Q \| P) + \ln \mathbb{E}_{h \sim P} [e^{\phi(h)}]. \quad (14.1)$$

Proof. With $\mathcal{H}_Q = \{h : Q(h) > 0\}$, we combine the two sums of the left-hand side minus the KL:

$$\begin{aligned} \mathbb{E}_{h \sim Q} [\phi(h)] - \text{KL}(Q \| P) &= \sum_{h \in \mathcal{H}_Q} Q(h) \left(\phi(h) - \ln \frac{Q(h)}{P(h)} \right) \\ &\downarrow \phi(h) = \ln e^{\phi(h)} \text{ and } \ln a - \ln b = \ln \frac{a}{b} \\ &= \sum_{h \in \mathcal{H}_Q} Q(h) \ln \frac{P(h) e^{\phi(h)}}{Q(h)} \\ &\downarrow \text{Jensen's inequality for the concave logarithm} \\ &\leq \ln \sum_{h \in \mathcal{H}_Q} Q(h) \frac{P(h) e^{\phi(h)}}{Q(h)} = \ln \sum_{h \in \mathcal{H}_Q} P(h) e^{\phi(h)} \\ &\downarrow \text{we add the non-negative terms of the voters with } Q(h) = 0 \\ &\leq \ln \sum_{h \in \mathcal{H}} P(h) e^{\phi(h)} = \ln \mathbb{E}_{h \sim P} [e^{\phi(h)}]. \quad \square \end{aligned}$$

In a nutshell Read (14.1) as follows: the average of ϕ under any posterior Q is controlled by a quantity computed under the prior, $\ln \mathbb{E}_P e^\phi$, plus a price, $\text{KL}(Q \| P)$, for having moved from P to Q . If $Q = P$ the price is 0; the further Q is from P , the more we pay. The right-hand side does not depend on Q except through the KL: this is what will allow the bound to hold for *all* posteriors at once.

14.3 A first PAC-Bayesian bound

We can now state and prove our first PAC-Bayesian bound. It contains a free parameter $\lambda > 0$, which must be fixed before seeing the data.

Theorem 14.4: A PAC-Bayesian bound (Catoni, 2007; Alquier, 2024)

Let P be a prior on $\mathcal{H}$ and $\lambda > 0$, both fixed before seeing the sample $S \sim \mathcal{D}^m$, and $\delta \in (0, 1]$. With probability at least $1 - \delta$ over S , for all distributions Q on $\mathcal{H}$,

$$R_{\mathcal{D}}(G_Q) \leq \hat{R}_S(G_Q) + \frac{\text{KL}(Q \| P) + \ln \frac{1}{\delta}}{\lambda} + \frac{\lambda}{8m}. \quad (14.2)$$

Proof. For each voter h , let $\Delta(h) = R_{\mathcal{D}}(h) - \hat{R}_S(h)$ be its *generalization gap*. The proof has three steps (Fig. 118).

Step 1: one voter at a time. Each voter h is fixed before seeing S (it belongs to $\mathcal{H}$, which is fixed in advance). By Hoeffding's lemma (13.1), applied to the errors of h ,

$$\mathbb{E}_S[e^{\lambda\Delta(h)}] \leq e^{\lambda^2/(8m)} \quad \text{for every } h \in \mathcal{H}.$$

Step 2: average over the prior, then Markov. Consider the random variable $Z = \mathbb{E}_{h \sim P}[e^{\lambda\Delta(h)}]$, which depends on the sample through the gaps $\Delta(h)$, but not through P , which is fixed. It is non-negative, and

$$\begin{aligned} \mathbb{E}_S[Z] &= \mathbb{E}_S\left[\sum_h P(h) e^{\lambda\Delta(h)}\right] \\ &\quad \downarrow \text{linearity of the expectation: } P \text{ does not depend on } S \\ &= \sum_h P(h) \mathbb{E}_S[e^{\lambda\Delta(h)}] \\ &\quad \downarrow \text{Step 1, and } \sum_h P(h) = 1 \\ &\leq e^{\lambda^2/(8m)}. \end{aligned}$$

By Markov's inequality (Lemma 13.1, second form), with probability at least $1 - \delta$ over S ,

$$\mathbb{E}_{h \sim P}[e^{\lambda\Delta(h)}] \leq \frac{e^{\lambda^2/(8m)}}{\delta}. \quad (14.3)$$

Step 3: change of measure. Assume that (14.3) holds (this is the case with probability at least $1 - \delta$). Then, for *every* distribution Q ,

$$\begin{aligned} \lambda(R_{\mathcal{D}}(G_Q) - \hat{R}_S(G_Q)) &= \mathbb{E}_{h \sim Q}[\lambda\Delta(h)] \\ &\quad \downarrow \text{change of measure (Lemma 14.3) with } \phi = \lambda\Delta \\ &\leq \text{KL}(Q\|P) + \ln \mathbb{E}_{h \sim P}[e^{\lambda\Delta(h)}] \\ &\quad \downarrow \text{inequality (14.3), and } \ln \frac{e^a}{\delta} = a + \ln \frac{1}{\delta} \\ &\leq \text{KL}(Q\|P) + \frac{\lambda^2}{8m} + \ln \frac{1}{\delta}. \end{aligned}$$

Dividing by $\lambda > 0$ and moving $\hat{R}_S(G_Q)$ to the right gives (14.2). Since (14.3) does not involve Q , the conclusion holds for all Q simultaneously. $\square$

Occam's bound is a special case. Take for Q a Dirac distribution δ_h on one voter h . Then $R_{\mathcal{D}}(G_Q) = R_{\mathcal{D}}(h)$, $\hat{R}_S(G_Q) = \hat{R}_S(h)$, and $\text{KL}(\delta_h\|P) = \ln \frac{1}{P(h)}$ (example 14.1): (14.2) becomes a bound of the same form as Occam's bound. The PAC-Bayesian bound extends it from one chosen classifier to any distribution over classifiers.

14.4 The bounds used in practice

Theorem 14.4 has a drawback: λ must be fixed before seeing the data. If we knew $\text{KL}(Q\|P)$ in advance, the best choice would be $\lambda = \sqrt{8m(\text{KL}(Q\|P) + \ln \frac{1}{\delta})}$, which

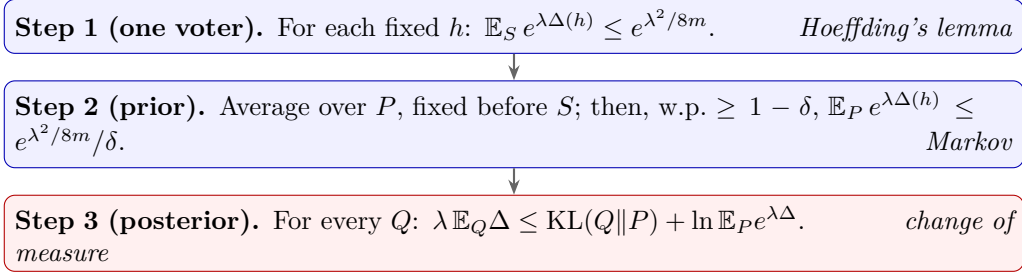


Figure 118: The three steps of the proof of Theorem 14.4. Only Step 2 involves the randomness of the sample, and it does not involve the posterior: this is why the bound holds for all posteriors at once. It is also the only step that requires the prior to be fixed before seeing the data.

balances the two last terms of (14.2) and gives a margin $\sqrt{(\text{KL}(Q\|P) + \ln \frac{1}{\delta})/(2m)}$. But the KL depends on Q , which depends on the data. Applying Theorem 14.4 for a whole grid of values of λ simultaneously (with a union bound), or slightly changing the function Δ in the proof, removes this difficulty at a very small cost, and leads to the two bounds used in practice. We state them without proof.

Theorem 14.5: McAllester's and Seeger's bounds (McAllester, 1999; Seeger, 2002; Maurer, 2004)

Let P be a prior on $\mathcal{H}$, fixed before seeing $S \sim \mathcal{D}^m$ ($m \geq 8$), and $\delta \in (0, 1]$. With probability at least $1 - \delta$ over S , for all distributions Q on $\mathcal{H}$, the two following bounds hold:

$$\text{(McAllester)} \quad R_{\mathcal{D}}(G_Q) \leq \hat{R}_S(G_Q) + \sqrt{\frac{\text{KL}(Q\|P) + \ln \frac{2\sqrt{m}}{\delta}}{2m}}, \quad (14.4)$$

$$\text{(Seeger)} \quad R_{\mathcal{D}}(G_Q) \leq \bar{\text{kl}}^{-1}\left(\hat{R}_S(G_Q), \frac{\text{KL}(Q\|P) + \ln \frac{2\sqrt{m}}{\delta}}{m}\right). \quad (14.5)$$

The two bounds are the PAC-Bayesian versions of the two test-set bounds of Section 13: Hoeffding's (margin $\sqrt{\cdot/2m}$) and the kl bound. As for a single classifier, Seeger's bound is always at least as good as McAllester's (by Pinsker's inequality), and much better when the empirical Gibbs risk is small. It is the bound we use for computing certificates; McAllester's bound is the one we use for *understanding*, because it has a simple formula. Compared with the test-set bound, only two things have changed: the term $\text{KL}(Q\|P)$, the price for having chosen Q with the data, and $\ln \frac{1}{\delta}$ replaced by $\ln \frac{2\sqrt{m}}{\delta}$, a small price for not having fixed λ .

14.5 Reading the bound

McAllester's bound (14.4) can be read like a sentence:

$$\underbrace{R_{\mathcal{D}}(G_Q)}_{\text{what we want}} \leq \underbrace{\hat{R}_S(G_Q)}_{\text{fit to the data}} + \sqrt{\frac{\underbrace{\text{KL}(Q\|P)}_{\text{distance to the prior}} + \underbrace{\ln \frac{2\sqrt{m}}{\delta}}_{\text{confidence}}}{\underbrace{2m}_{\text{sample size}}}}.$$

- **The sample size m :** the price decreases like $1/\sqrt{m}$.
- **The confidence δ** only enters through a logarithm: going from 95% to 99.9% confidence costs little.
- **The KL** is the price for learning: a posterior that stays close to the prior is cheap, a posterior that concentrates on a few voters is expensive.
- **The size of $\mathcal{H}$** does not appear: only the distance between Q and P matters. This is what makes the bound usable for large ensembles.

A trade-off. Concentrating Q on the voters with the smallest empirical risks decreases the first term, but increases the KL. The best posterior realizes a compromise (Fig. 119).

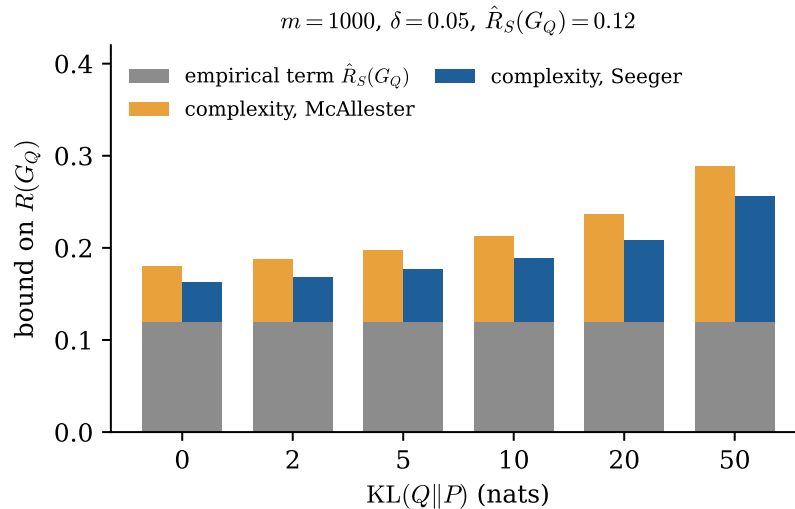


Figure 119: Anatomy of the bound, for $m = 1000$, $\delta = 0.05$ and an empirical Gibbs risk of 0.12. Grey: the empirical risk; orange and blue: the price added by McAllester's and by Seeger's bound, as $\text{KL}(Q\|P)$ grows from 0 to 50. Seeger's bound is always smaller.

[code: make_figures.py: fig_bound_anatomy]

Example 14.2 (Computing the bounds). Take $m = 1000$, $\delta = 0.05$ and an empirical Gibbs risk $\hat{R}_S(G_Q) = 0.12$. Then $\ln \frac{2\sqrt{m}}{\delta} = \ln \frac{2 \times 31.6}{0.05} \approx 7.14$, and:

KL($Q\ P$)	0	2.3	4.6	20	50
McAllester	0.180	0.189	0.197	0.236	0.289
Seeger	0.162	0.169	0.176	0.209	0.256

For McAllester with $KL = 0$, for instance: $0.12 + \sqrt{7.14/2000} = 0.12 + 0.060 = 0.180$. The values 2.3 and 4.6 are those of a posterior uniform on 10 voters out of 100, and of a single voter out of 100 (example 14.1). Even with a KL of 50, which corresponds to a posterior very far from the prior, the bound remains far below 1.

Running example Let us certify the Gibbs classifier of our forest (a tree drawn at random), with the uniform prior and the uniform posterior, so that $KL(Q||P) = 0$. We need examples that the trees have not been trained on. For now, let us admit that each tree can be evaluated on the examples left out of its bootstrap sample, its *out-of-bag* examples (Section 16.2). The average out-of-bag error of the trees is 0.174, computed with at least 747 examples per tree. With $\delta = 0.05$, McAllester's bound gives 0.243 and Seeger's bound 0.230. The test Gibbs risk, which the bound did not see, is 0.190: the certificate is valid and not far from the truth, and it was obtained without a test set. But the vote has a test risk of 0.044, and nothing in this section says anything about it: this is the object of Section 15.

14.6 Choosing the prior and the posterior

The prior. Any prior gives a valid bound, provided it is fixed before seeing the data. The uniform prior is the default choice. A better prior gives a smaller KL for the posteriors we are likely to learn, hence a tighter bound. A classical trick to obtain a good prior without breaking the rules is to *split the sample*: a first part of the data is used to build the prior (for instance to pre-train a model), and the bound is computed on the second part only, which the prior has not seen.

The posterior. The bound holds for all posteriors at once, so we are free to choose the posterior that makes it as small as possible. For the bound (14.2) with λ fixed, the answer has a closed form.

Proposition 14.6: The Gibbs posterior (Catoni, 2007)

For $\lambda > 0$, the distribution that minimizes $\hat{R}_S(G_Q) + \frac{KL(Q||P)}{\lambda}$, hence the right-hand side of (14.2), is

$$Q_\lambda(h) = \frac{P(h) e^{-\lambda \hat{R}_S(h)}}{Z_\lambda}, \quad Z_\lambda = \sum_{h' \in \mathcal{H}} P(h') e^{-\lambda \hat{R}_S(h')}. \quad (14.6)$$

Proof. For any Q , we write λ times the objective as a single sum:

$$\begin{aligned} \lambda \hat{R}_S(G_Q) + KL(Q||P) &= \sum_h Q(h) \left(\lambda \hat{R}_S(h) + \ln \frac{Q(h)}{P(h)} \right) \\ &\downarrow \lambda \hat{R}_S(h) = -\ln e^{-\lambda \hat{R}_S(h)} \\ &= \sum_h Q(h) \ln \frac{Q(h)}{P(h) e^{-\lambda \hat{R}_S(h)}} \\ &\downarrow \text{definition of } Q_\lambda: P(h) e^{-\lambda \hat{R}_S(h)} = Z_\lambda Q_\lambda(h) \end{aligned}$$

$$= \sum_h Q(h) \ln \frac{Q(h)}{Q_\lambda(h)} - \ln Z_\lambda = \text{KL}(Q \| Q_\lambda) - \ln Z_\lambda.$$

The last term does not depend on Q , and $\text{KL}(Q \| Q_\lambda) \geq 0$ with equality if and only if $Q = Q_\lambda$ (Proposition 14.2). $\square$

The Gibbs posterior reweights the prior: each voter keeps its prior weight, multiplied by $e^{-\lambda \hat{R}_S(h)}$, a factor that is larger for voters with a small empirical risk. The parameter λ plays the role of an inverse temperature (Fig. 120): for $\lambda = 0$ we keep the prior, and for $\lambda \rightarrow \infty$ all the weight goes to the voter with the smallest empirical risk, the empirical risk minimizer. In between, the Gibbs posterior is a *soft* version of empirical risk minimization, which spreads its weight over all the good voters. When the loss is the negative log-likelihood and $\lambda = m$, it is exactly the posterior of Bayesian statistics: this is where the name *PAC-Bayes* comes from.

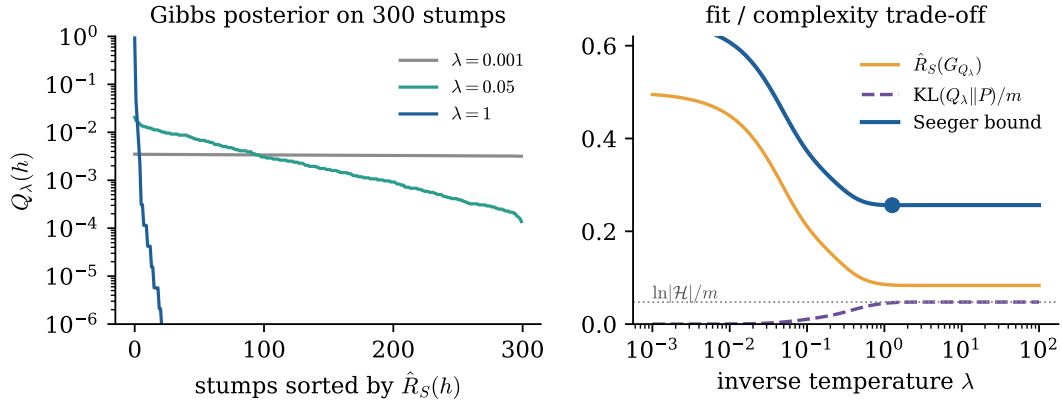


Figure 120: The Gibbs posterior on 300 decision stumps, with $m = 120$ examples. Left: the weights $Q_\lambda(h)$ of the stumps, sorted by empirical risk, for three values of λ (in the figure, λ is given per example, i.e. divided by m : the values 0.001, 0.05 and 1 correspond to $\lambda = 0.12, 6$ and 120 in (14.6)). Right: empirical Gibbs risk, KL divergence and Seeger's bound as functions of λ ; the bound is minimal for an intermediate value of λ , which spreads the weight over the good stumps. [code: make_figures.py: fig_temperature]

To remember from Section 14

- The *KL divergence* $\text{KL}(Q \| P) = \sum_h Q(h) \ln \frac{Q(h)}{P(h)}$ measures how far the posterior is from the prior; it is ≥ 0 , equal to 0 iff $Q = P$, and equal to $\ln \frac{1}{P(h)}$ for a Dirac posterior.
- The *change of measure* $\mathbb{E}_Q \phi \leq \text{KL}(Q \| P) + \ln \mathbb{E}_P e^\phi$ transfers a statement on the prior to every posterior.
- *PAC-Bayesian bound*: with probability $1 - \delta$, for all Q , $R_{\mathcal{D}}(G_Q) \leq \hat{R}_S(G_Q) + \frac{\text{KL}(Q \| P) + \ln(1/\delta)}{\lambda} + \frac{\lambda}{8m}$. Proof: Hoeffding for each voter, Markov on the prior average, change of measure.
- In practice: McAllester's bound (simple formula) and Seeger's bound (tighter, computed with $\overline{\text{kl}}^{-1}$), with $\ln \frac{2\sqrt{m}}{\delta}$ instead of $\ln \frac{1}{\delta}$.
- Concentrating Q lowers the empirical risk but raises the KL. The *Gibbs*

posterior $Q_\lambda \propto P e^{-\lambda \hat{R}_S}$ realizes the best compromise.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

- Q14.1.** P is uniform on 20 voters and Q is uniform on 5 of them. What is $\text{KL}(Q\|P)$?
- Q14.2.** What is $\text{KL}(Q\|P)$ when Q gives a positive weight to a voter h with $P(h) = 0$?
- Q14.3.** Which step of the proof of Theorem 14.4 requires the prior to be independent of the sample?
- Q14.4.** Why does the bound hold for all posteriors at the same time?
- Q14.5.** Compute McAllester's bound for $m = 2000$, $\delta = 0.05$, $\hat{R}_S(G_Q) = 0.2$ and $\text{KL}(Q\|P) = 0$.
- Q14.6.** What does the Gibbs posterior become when $\lambda = 0$? When λ is very large?

15 From the Gibbs classifier to the majority vote

About this section

Guiding question. *Why is a vote better than its average voter, and how can we certify the vote itself?*

Section 14 certifies the Gibbs classifier, a voter drawn at random. But we predict with the majority vote, which can be much better. We introduce one random variable, the weight W_Q of the voters that err on an example, which describes both predictors. Markov's inequality applied to W_Q gives the first-order bound: the vote is at most twice worse than the Gibbs classifier. Applied to W_Q^2 , it gives the tandem bound, which involves pairs of voters and sees their diversity. This answers the explanatory question of Section 12.

Throughout the section, the voters take values in $\{-1, +1\}$, Q is a distribution over a finite set $\mathcal{H}$ of voters, and the vote $B_Q(x) = \text{sign}(\sum_h Q(h)h(x))$ is counted as wrong in case of a tie.

15.1 One random variable that describes the vote

Definition 15.1: Weight of the erring voters

For an example (x, y) , the **weight of the erring voters** is

$$W_Q(x, y) = \sum_{h \in \mathcal{H}} Q(h) \mathbb{1}[h(x) \neq y] \in [0, 1],$$

that is, the proportion of voters (weighted by Q) that err on (x, y) .

When (x, y) is drawn from $\mathcal{D}$, $W_Q(x, y)$ is a random variable with values in $[0, 1]$. Both predictors can be read on it (Fig. 121).

- **The Gibbs risk is its mean.** Exchanging the sum over the voters and the expectation over the data,

$$\begin{aligned} \mathbb{E}_{(x,y) \sim \mathcal{D}} [W_Q(x, y)] &= \mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\sum_h Q(h) \mathbb{1}[h(x) \neq y] \right] \\ &\downarrow \text{linearity of the expectation} \\ &= \sum_h Q(h) \mathbb{P}_{(x,y) \sim \mathcal{D}}(h(x) \neq y) = \sum_h Q(h) R_{\mathcal{D}}(h) = R_{\mathcal{D}}(G_Q). \end{aligned} \tag{15.1}$$

- **The vote errs when $W_Q \geq \frac{1}{2}$.** The vote errs on (x, y) when the voters that err weigh at least as much as those that are right, that is when $W_Q(x, y) \geq \frac{1}{2}$. Hence

$$R_{\mathcal{D}}(B_Q) = \mathbb{P}_{(x,y) \sim \mathcal{D}} \left(W_Q(x, y) \geq \frac{1}{2} \right). \tag{15.2}$$

In a nutshell The Gibbs risk only depends on the *mean* of W_Q , whereas the risk of the vote depends on how often W_Q exceeds $\frac{1}{2}$, that is on the *tail* of

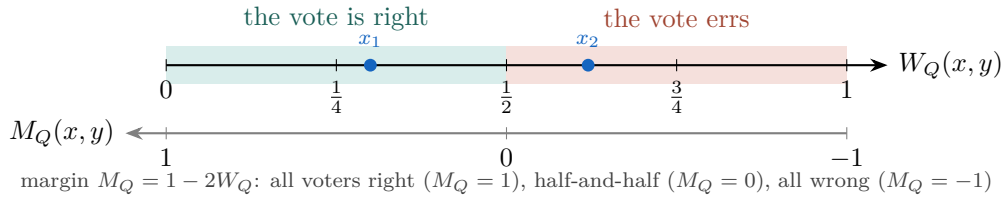


Figure 121: The weight W_Q of the erring voters on an example. On x_1 , 30% of the voters (weighted by Q) err and the vote is right; on x_2 , 62% of them err and the vote errs. The *margin* $M_Q = 1 - 2W_Q$ (Section 15.6) measures the same thing on the scale $[-1, 1]$. [code: figures/tikz/wq_line.tex]

its distribution. Two ensembles with the same Gibbs risk can have very different votes (Fig. 122). Bounding a tail with a mean is exactly what Markov's inequality does.

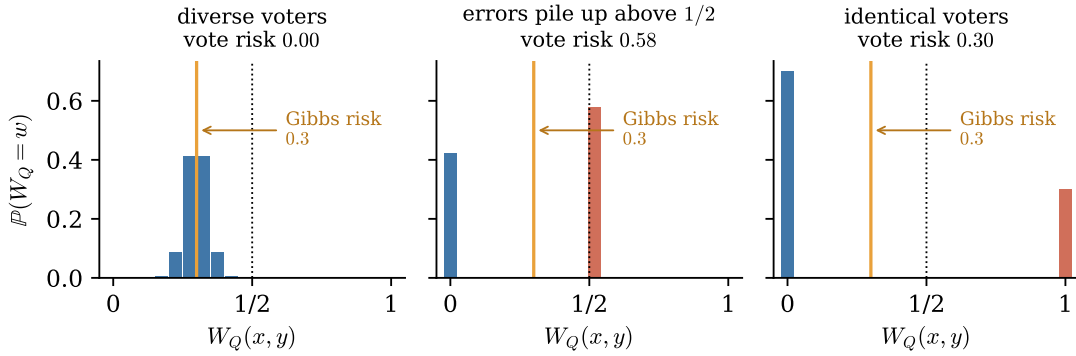


Figure 122: Three distributions of W_Q with the same mean, i.e. the same Gibbs risk 0.3. Left: the voters are diverse and their errors never pile up, so the vote is always right. Middle: the errors pile up just above $\frac{1}{2}$ on 58% of the examples, where the vote errs; this is close to the worst case of the first-order bound. Right: identical voters, the vote is as good as one voter. [code: make_figures.py: fig_wq_scenarios]

15.2 The first-order bound

Proposition 15.2: First-order bound (Langford and Shawe-Taylor, 2002)

For every distribution Q , $R_{\mathcal{D}}(B_Q) \leq 2 R_{\mathcal{D}}(G_Q)$.

Proof. W_Q is non-negative, so Markov's inequality (Lemma 13.1) applies:

$$\begin{aligned}
 R_{\mathcal{D}}(B_Q) &= \mathbb{P}\left(W_Q \geq \frac{1}{2}\right) \\
 &\downarrow \text{Markov's inequality with } a = \frac{1}{2} \\
 &\leq \frac{\mathbb{E} W_Q}{1/2} \\
 &\downarrow \text{the mean of } W_Q \text{ is the Gibbs risk (15.1)} \\
 &= 2 R_{\mathcal{D}}(G_Q).
 \end{aligned}$$

□

Combined with Seeger's bound on the Gibbs risk, it gives a certificate for the vote: with probability at least $1 - \delta$, for all Q ,

$$R_{\mathcal{D}}(B_Q) \leq 2 \bar{\text{kl}}^{-1} \left(\hat{R}_S(G_Q), \frac{\text{KL}(Q \| P) + \ln \frac{2\sqrt{m}}{\delta}}{m} \right). \quad (15.3)$$

When is it tight? Markov's inequality is an equality when the variable only takes the values 0 and a . Here: on each example, either all the voters are right ($W_Q = 0$), or exactly half of them err ($W_Q = \frac{1}{2}$), and the vote errs on all the examples of the second kind (the middle panel of Fig. 122 is close to this situation). This is the worst case: the voters pile up their errors on the same examples, just enough to make the vote err. In general the bound is loose. It cannot explain why a vote is *better* than its voters, since it is always above the Gibbs risk: in the running example, it gives $2 \times 0.19 = 0.38$ for a vote whose test risk is 0.044.

15.3 Condorcet's jury theorem

The opposite of the worst case is a vote of *independent* voters. This situation was already studied by Condorcet (1785) for juries.

Proposition 15.3: Condorcet's jury theorem

Let $h_1, \dots, h_n$ be n voters, with uniform weights, that err independently of each other, each with probability $p < \frac{1}{2}$ on a random example. Then

$$R_{\mathcal{D}}(B_Q) \leq e^{-2n(\frac{1}{2}-p)^2} \xrightarrow{n \rightarrow \infty} 0.$$

Proof. On a random example, $W_Q = \frac{1}{n} \sum_{t=1}^n \mathbb{1}[h_t(x) \neq y]$ is an average of n independent Bernoulli variables of mean p . By Hoeffding's inequality (Theorem 13.2 applied to the variables $1 - \mathbb{1}[h_t(x) \neq y]$, which gives the same bound for the upper tail),

$$R_{\mathcal{D}}(B_Q) = \mathbb{P}\left(W_Q \geq \frac{1}{2}\right) = \mathbb{P}\left(W_Q - p \geq \frac{1}{2} - p\right) \leq e^{-2n(\frac{1}{2}-p)^2}. \quad \square$$

With $n = 101$ voters of error 0.3, the vote errs with probability less than $e^{-8.1} < 0.0004$. Independent voters are much better together than alone. Of course, voters learned on the same data are never independent: their errors are *correlated*, they tend to err on the same difficult examples, and the gain of the vote is limited (Fig. 123, right). What matters is how much the voters err *together*.

15.4 Pairs of voters: joint error and disagreement

To measure how much the voters err together, we look at *pairs* of voters (h, h') drawn independently from Q .

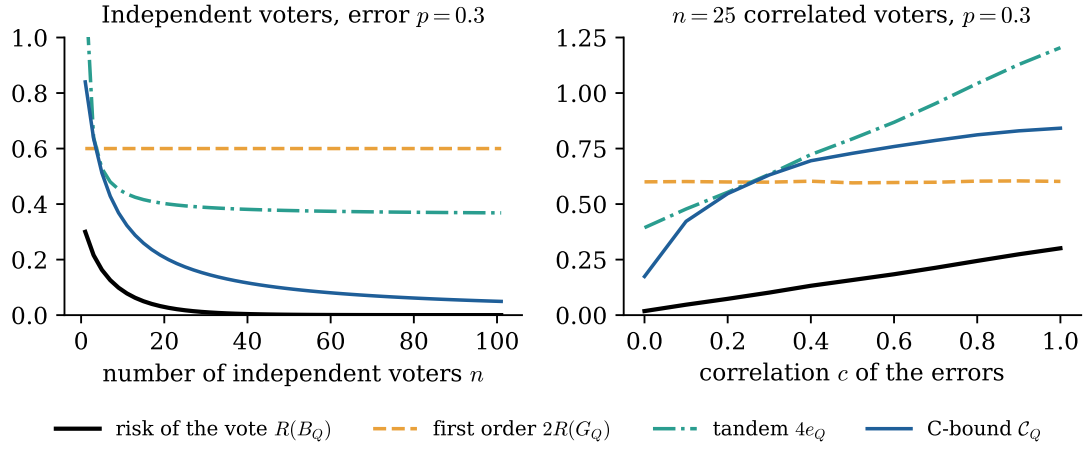


Figure 123: Risk of the majority vote (black) and of the bounds of this section, for voters of individual error $p = 0.3$. Left: independent errors, as a function of the number of voters; the first-order bound stays at $2p = 0.6$. Right: 25 voters whose errors are correlated (each voter copies the errors of a common leader with probability c); the more correlated the voters, the worse the vote. [code: make_figures.py: fig_condorcet]

Definition 15.4: Joint error and disagreement

The **joint error** $e_{\mathcal{D}}(Q)$ is the probability that two voters drawn from Q both err on a random example, and the **disagreement** $d_{\mathcal{D}}(Q)$ the probability that they give different answers:

$$e_{\mathcal{D}}(Q) = \sum_{h, h'} Q(h)Q(h') \mathbb{P}(h(x) \neq y \text{ and } h'(x) \neq y),$$

$$d_{\mathcal{D}}(Q) = \sum_{h, h'} Q(h)Q(h') \mathbb{P}(h(x) \neq h'(x)).$$

The disagreement measures the *diversity* of the ensemble; it does not need the labels and can be computed on unlabelled data. The three quantities are linked by a simple identity (Fig. 124).

Proposition 15.5: Gibbs risk, joint error and disagreement

For every distribution Q , $R_{\mathcal{D}}(G_Q) = e_{\mathcal{D}}(Q) + \frac{1}{2} d_{\mathcal{D}}(Q)$.

Proof. Fix two voters h, h' and an example (x, y) . With binary labels there are three cases: both are right, both err, or exactly one errs, and exactly one errs if and only if they disagree. Counting the errors in each case,

$$\mathbb{1}[h(x) \neq y] + \mathbb{1}[h'(x) \neq y] = 2\mathbb{1}[h(x) \neq y, h'(x) \neq y] + \mathbb{1}[h(x) \neq h'(x)].$$

Take the expectation over $(x, y) \sim \mathcal{D}$ and over $h, h' \sim Q$. The left-hand side gives $2R_{\mathcal{D}}(G_Q)$, the right-hand side $2e_{\mathcal{D}}(Q) + d_{\mathcal{D}}(Q)$. Divide by 2. $\square$

	$h'(x) = y$	$h'(x) \neq y$	
$h(x) = y$	both correct	disagree h right, h' wrong	$h, h' \sim Q$ independent, $(x, y) \sim \mathcal{D}$: ■ joint error: $e_{\mathcal{D}}(Q) = \mathbb{P}(\text{red})$ ■ disagreement: $d_{\mathcal{D}}(Q) = \mathbb{P}(\text{orange})$ Gibbs risk: red + one orange cell, $R_{\mathcal{D}}(G_Q) = e_{\mathcal{D}}(Q) + \frac{1}{2}d_{\mathcal{D}}(Q)$.
$h(x) \neq y$	disagree h wrong, h' right	both wrong (tandem loss)	

Figure 124: Two voters h, h' drawn from Q on an example: the four cases. The joint error is the probability of the red cell, the disagreement that of the two orange cells. The Gibbs risk is the red cell plus one orange cell. [code: figures/tikz/tandem_table.tex]

In a nutshell For a given Gibbs risk (a given quality of the voters), the more the voters disagree, the less often they err together. This is diversity: *good voters that do not make the same mistakes*. The next bound shows that the joint error, not the Gibbs risk, controls the vote.

15.5 The tandem bound

Proposition 15.6: Tandem bound (Masegosa et al., 2020)

For every distribution Q , $R_{\mathcal{D}}(B_Q) \leq 4e_{\mathcal{D}}(Q)$.

Proof. Since $W_Q \geq 0$, $W_Q \geq \frac{1}{2}$ if and only if $W_Q^2 \geq \frac{1}{4}$. Hence

$$\begin{aligned}
 R_{\mathcal{D}}(B_Q) &= \mathbb{P}\left(W_Q^2 \geq \frac{1}{4}\right) \\
 &\downarrow \text{Markov's inequality applied to } W_Q^2 \geq 0, \text{ with } a = \frac{1}{4} \\
 &\leq 4\mathbb{E}[W_Q^2].
 \end{aligned}$$

It remains to compute $\mathbb{E}[W_Q^2]$. Developing the square of the sum,

$$\begin{aligned}
 \mathbb{E}[W_Q^2] &= \mathbb{E}\left[\sum_h Q(h)\mathbb{1}[h(x) \neq y] \sum_{h'} Q(h')\mathbb{1}[h'(x) \neq y]\right] \\
 &\downarrow \text{product of indicators: both err} \\
 &= \sum_{h, h'} Q(h)Q(h') \mathbb{P}(h(x) \neq y, h'(x) \neq y) = e_{\mathcal{D}}(Q). \quad \square
 \end{aligned}$$

The tandem bound uses the *second moment* of W_Q , where the first-order bound only used its mean: it sees whether the errors pile up. When the voters are diverse, $e_{\mathcal{D}}(Q)$ can be much smaller than $R_{\mathcal{D}}(G_Q)/2$, and the tandem bound much better than the first-order bound. When they are not, it can be worse than the first-order bound: for identical voters, $e_{\mathcal{D}}(Q) = R_{\mathcal{D}}(G_Q)$ and the tandem bound is $4R_{\mathcal{D}}(G_Q)$.

Example 15.1 (Three voters, again). In Example 12.1, three voters with weights $\frac{1}{3}$ each err on a different point among four. A voter paired with itself errs with probability 0.25; two different voters never err together. Among the 9 pairs (h, h') , 3 are a voter with itself, so $e_{\mathcal{D}}(Q) = \frac{3}{9} \times 0.25 \approx 0.083$. Two different voters disagree on 2 points out of 4,

so $d_{\mathcal{D}}(Q) = \frac{6}{9} \times 0.5 \approx 0.333$. We check the identity: $0.083 + \frac{1}{2} \times 0.333 = 0.25 = R_{\mathcal{D}}(G_Q)$. The first-order bound gives 0.5, the tandem bound $4 \times 0.083 = 0.333$; the true risk of the vote is 0.

The C-bound. An even finer bound, the *C-bound* (Lacasse et al., 2006; Germain et al., 2015), uses both the mean and the variance of W_Q (through Chebyshev–Cantelli’s inequality): if $R_{\mathcal{D}}(G_Q) < \frac{1}{2}$,

$$R_{\mathcal{D}}(B_Q) \leq 1 - \frac{(1 - 2R_{\mathcal{D}}(G_Q))^2}{1 - 2d_{\mathcal{D}}(Q)}.$$

In the three-voter example, it gives $1 - \frac{0.25}{0.333} = 0.25$. We will not use it further: it is harder to certify from data, because it needs two estimates at once.

Certifying the vote. The joint error is the Gibbs risk of a “pair” of voters, for the loss “both err”, with the distribution Q^2 over pairs. Seeger’s bound therefore applies to it, with the empirical joint error $\hat{e}_S(Q)$ (the proportion of pairs that both err on the sample, averaged with Q^2) and $\text{KL}(Q^2 \| P^2) = 2 \text{KL}(Q \| P)$. With probability at least $1 - \delta$, for all Q ,

$$R_{\mathcal{D}}(B_Q) \leq 4 \bar{\text{kl}}^{-1} \left(\hat{e}_S(Q), \frac{2 \text{KL}(Q \| P) + \ln \frac{2\sqrt{m}}{\delta}}{m} \right). \quad (15.4)$$

Running example We can now answer the explanatory question of Section 12.

On the test set, the trees of our forest have a Gibbs risk of 0.190, a disagreement of 0.266 and a joint error of 0.057 (check: $0.057 + 0.266/2 = 0.190$). Two trees drawn at random disagree on more than a quarter of the digits, but both err on only 5.7% of them. The first-order bound is 0.381, the tandem bound $4 \times 0.057 = 0.229$, and the C-bound 0.179, for a vote whose risk is 0.044. *The vote is good because the trees are diverse*: they are individually mediocre, but their errors are spread over different digits.

Without a test set, estimating the same quantities on the out-of-bag examples (Section 16.2) with $\delta = 0.05$ certifies the vote at 0.460 with the first-order bound (15.3) and at 0.387 with the tandem bound (15.4). These certificates are valid, but far from the test risk: the factor 4 and the doubled KL are expensive. Section 16 improves them by learning the weights of the trees.

15.6 Margins and boosting

The quantity W_Q is closely related to the *margin* of the vote, met with AdaBoost in Section 10.2.2:

$$M_Q(x, y) = y \sum_h Q(h) h(x) = 1 - 2W_Q(x, y) \in [-1, 1].$$

(Each voter contributes $+Q(h)$ when it is right and $-Q(h)$ when it errs.) The vote errs when $M_Q \leq 0$, and the margin is close to 1 when almost all the voters agree on the right answer. The bounds of this section are statements about the distribution of the

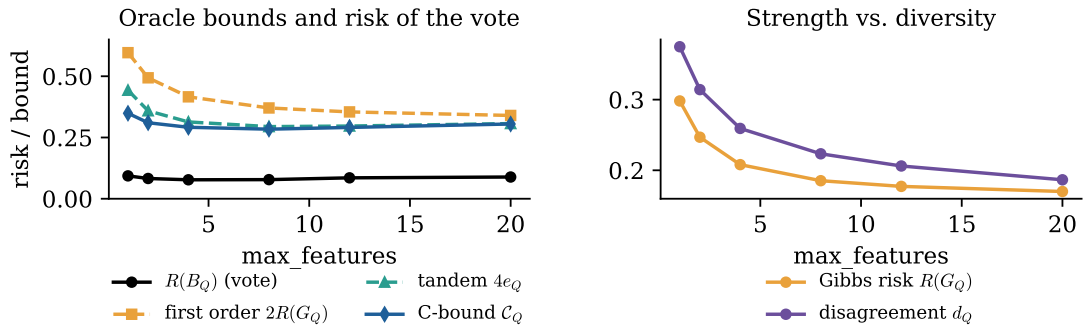


Figure 125: Random forests of 100 trees on a simulated problem, for an increasing number of features tried at each split. Left: risk of the vote and oracle values of the bounds, computed on a large test sample. Right: stronger trees (small Gibbs risk) are also less diverse (small disagreement). The tandem bound and the C-bound follow the risk of the vote much better than the first-order bound. [code: make_figures.py: fig_mv_bounds_diversity]

margin: the first-order bound uses its mean, the tandem bound its second moment. This explains the observation of [Schapire et al. \(1998\)](#) recalled in [Section 10.2.2](#): AdaBoost keeps increasing the margins of the training examples after its training error has reached 0 ([Fig. 126](#)); pushing the distribution of the margins away from 0 makes W_Q small on most examples, hence a small joint error and a good vote.

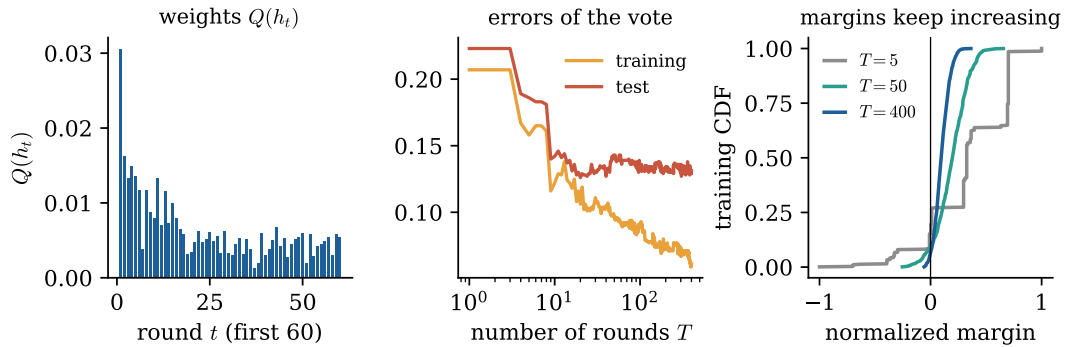


Figure 126: AdaBoost with decision stumps on a simulated problem. Left: the normalized weights $Q(h_t)$ of the first stumps form a distribution over the weak learners. Middle: the training error of the vote goes to zero while the test error stabilizes. Right: the distribution of the margins keeps moving to the right. [code: make_figures.py: fig_adaboost]

To remember from Section 15

- The *weight of the erring voters* W_Q describes both predictors: the Gibbs risk is its mean, the vote errs when $W_Q \geq \frac{1}{2}$.
- *First-order bound*: $R_{\mathcal{D}}(B_Q) \leq 2R_{\mathcal{D}}(G_Q)$ (Markov on W_Q). Simple, but it cannot explain why a vote beats its voters.
- *Condorcet*: independent voters better than chance make an almost perfect vote. Real voters are correlated.
- *Joint error* $e_{\mathcal{D}}$ and *disagreement* $d_{\mathcal{D}}$: $R_{\mathcal{D}}(G_Q) = e_{\mathcal{D}}(Q) + \frac{1}{2}d_{\mathcal{D}}(Q)$. For a given Gibbs risk, more diversity means a smaller joint error.

- *Tandem bound:* $R_{\mathcal{D}}(B_Q) \leq 4e_{\mathcal{D}}(Q)$ (Markov on W_Q^2). Both bounds can be certified from data with Seeger's bound.
- A vote is good when its voters are good *and* diverse.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

- Q15.1.** On an example, voters weighing 0.7 in total are right. What is W_Q ? Is the vote right?
- Q15.2.** The Gibbs risk of an ensemble is 0.1. What does the first-order bound say about the vote?
- Q15.3.** An ensemble has a Gibbs risk of 0.2 and a disagreement of 0.3. What are its joint error and its tandem bound?
- Q15.4.** What are the joint error and the disagreement of an ensemble of identical voters of risk p ?
- Q15.5.** Why is the KL doubled in the certificate of the tandem bound?
- Q15.6.** How are the margin and W_Q related?

16 Learning with the bounds

About this section

Guiding question. *Since the bounds hold for every posterior, why not choose the posterior that minimizes them?*

So far, the bounds certified a posterior given by some algorithm. This section reverses the roles: we learn the posterior by minimizing a bound, and the minimal value of the bound is a guarantee on the learned model. We explain why this is legitimate, and which rule must not be broken. We then apply the idea to random forests, where the out-of-bag examples allow to learn and certify with the same data, and we see why the bound to minimize must take the diversity of the voters into account. A short tour of other applications closes the section; [Section 17](#) then derives one of these algorithms in detail.

16.1 From empirical risk minimization to bound minimization

In [Parts II](#) and [III](#), learning meant minimizing an empirical risk, possibly plus a regularization term, and the quality of the learned model was then estimated on a test set. A PAC-Bayesian bound suggests another way:

$$\hat{Q} \in \arg \min_Q \underbrace{\hat{R}_S(G_Q) + \text{price}(\text{KL}(Q\|P), m, \delta)}_{\text{the bound}}, \quad (16.1)$$

that is, minimize an *upper bound on the true risk* instead of the empirical risk. The first term asks for a posterior that fits the data, the second one for a posterior that stays close to the prior: the KL plays the role of a regularization, but with a weight that is dictated by the theory instead of being tuned.

The certificate comes for free. The bound holds, with probability at least $1 - \delta$, for *all* posteriors at once. It therefore holds for the posterior $\hat{Q}$ found by the minimization, whatever the algorithm used to find it (exact formula, gradient descent, grid search...). The value of the bound at $\hat{Q}$ is a valid guarantee on its true risk. Such a procedure, which outputs a model together with a guarantee on its risk, is called *self-bounding* ([Freund, 1998](#); [Viallard et al., 2021](#)).

Example 16.1 (The Gibbs posterior as a self-bounding algorithm). Take a finite set of voters (for instance 300 decision stumps built on a first sample), the uniform prior, and a second sample of m examples to compute their empirical risks. For each λ on a grid, compute the Gibbs posterior $Q_\lambda \propto e^{-\lambda \hat{R}_S(h)}$ ([Proposition 14.6](#)) and its Seeger bound ([14.5](#)); keep the λ with the smallest bound ([Fig. 120](#)). Choosing λ in this way is allowed: Seeger's bound holds for every posterior, so it holds for every Q_λ , whichever λ we keep. On the **breast cancer** dataset, with $m = 120$ examples, this gives a certificate of 0.256 on the risk of the Gibbs classifier, whose test risk is 0.114.

Pitfall Everything may depend on the data *except* the prior and the confidence δ . If the prior, or the set of voters on which it is defined, is chosen with the same examples as those used to compute the empirical risks in the bound, the bound is no longer valid. In particular, one must not try several priors on the same data and keep the one that gives the smallest bound (unless the price of this choice is

paid with a union bound). Splitting the sample, one part for the prior and the voters, the other part for the bound, is the safe solution.

16.2 Random forests and the out-of-bag examples

Splitting the sample wastes data: the examples used to build the voters are not used to certify them. Bagging offers a way out. Each tree h_t of a forest is learned on a bootstrap sample S_t , and the examples left out of it, its *out-of-bag* (OOB) examples $T_t = S \setminus S_t$, have never been seen by h_t (Fig. 127). From the point of view of the tree h_t , they are a fresh test sample. With bootstrap samples of size m , about $e^{-1} \approx 37\%$ of the sample is out-of-bag for each tree (Section 10.2.1); with bootstrap samples of size $m/2$, as in our running example, about $e^{-1/2} \approx 61\%$.

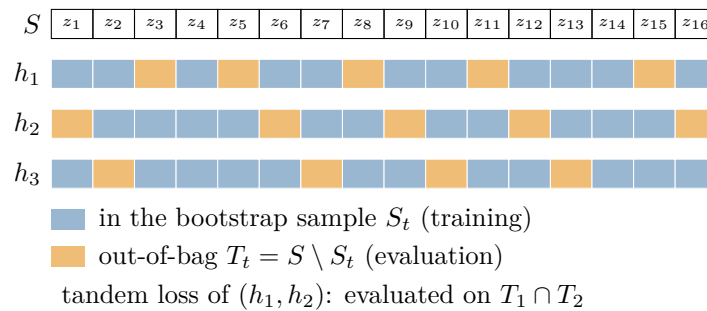


Figure 127: Out-of-bag sets of three trees. Blue: the examples of the bootstrap sample of a tree (used to train it); orange: its out-of-bag examples (used to evaluate it). The tandem loss of a pair of trees is evaluated on the examples that are out-of-bag for both. [code: figures/tikz/oob_sets.tex]

Thiemann et al. (2017) and Lorenzen et al. (2019) showed that this is enough to apply the bounds of Sections 14 and 15. The trick is to put the prior on the *indices* $1, \dots, n$ of the trees (the uniform prior, for instance), which are fixed before the forest is grown, and to evaluate each tree on its own out-of-bag examples:

- the empirical risk of a tree is its OOB error, and the empirical Gibbs risk the weighted average of these errors;
- the empirical joint error of a pair of trees is computed on the examples that are out-of-bag for both;
- the sample size m in the bounds is replaced by the size of the smallest of these OOB sets.

With these replacements, the first-order and the tandem certificates (15.3)–(15.4) remain valid. Every example is used both to grow some trees and to certify the others, and no test set is needed.

16.3 Learning the weights of a forest

We can now learn new weights Q for the 100 trees of our forest by minimizing one of the two certificates on the OOB data. Table 8 and Fig. 128 compare the results with the uniform weights of the forest.

Table 8: Three posteriors on the forest of 100 trees on **digits**: test risks of the vote and of the Gibbs classifier, OOB certificates on the risk of the vote ($\delta = 0.05$), and effective number of trees $1/\sum_t Q(h_t)^2$.

posterior	vote (test)	Gibbs (test)	first-order certificate	tandem certificate	effective trees
uniform (the forest)	0.044	0.190	0.460	0.387	100
minimizing the first-order bound	0.150	0.158	0.386	0.666	2
minimizing the tandem bound	0.043	0.183	0.436	0.361	58

Minimizing the first-order bound destroys the vote. The first-order bound only depends on the Gibbs risk, the average error of the trees. Its minimizer therefore puts almost all the weight on the trees with the smallest OOB error: only two effective trees remain. The Gibbs risk decreases (0.158), but the vote of two similar trees has lost all the benefit of diversity, and its risk jumps from 0.044 to 0.150.

Minimizing the tandem bound keeps the vote good. The tandem bound depends on the joint error of *pairs* of trees: putting a lot of weight on two trees that err on the same examples is expensive, spreading the weight over trees that err on different examples is cheap. Its minimizer keeps 58 effective trees, its vote is as good as the forest (0.043), and it obtains the best certificate, 0.361.

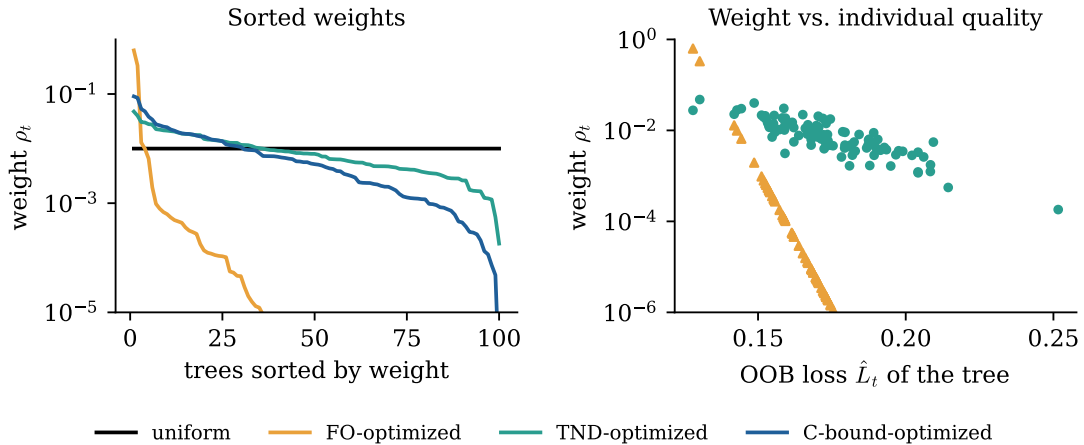


Figure 128: Weights learned on the forest of 100 trees on **digits**. Left: sorted weights of the posteriors (log scale). Right: weight of each tree as a function of its OOB error, for the first-order and the tandem posteriors: the first-order posterior selects the two best trees, the tandem posterior spreads its weight over many good trees.

[code: make_figures.py: fig_fo_iterations_and_weights]

In a nutshell A good learning objective for a vote must reward the diversity of the voters, not only their individual quality. The first-order bound does not, and its minimization leads to a bad vote; the tandem bound does. This is the lesson of [Section 15](#) turned into an algorithm.

Running example Let us gather what we have learned about our forest of 100 trees on digits. Its vote errs on 4.4% of the test digits, while an average tree errs on 19% of them; the gap is explained by the diversity of the trees, whose disagreement (0.27) exceeds their Gibbs risk (Section 15). Setting aside the 540 test digits certifies the vote at 0.069 (Section 13). Without any test set, the OOB tandem bound certifies the forest at 0.387, and learning the weights by minimizing it lowers the certificate to 0.361, with an equally good vote. These certificates are valid but still far from the test risk: certifying a vote from the training data alone remains harder than evaluating it on a test set. Their advantage is that every example has been used for learning.

16.4 Beyond ensembles

The same recipe applies to many other models; we only give two examples.

Linear classifiers. Take as voters all the linear classifiers $h_{\mathbf{v}}(x) = \text{sign}(\langle \mathbf{v}, x \rangle)$, as prior a standard Gaussian distribution $P = \mathcal{N}(\mathbf{0}, I)$ on $\mathbf{v}$, and as posteriors the Gaussian distributions $Q = \mathcal{N}(\mathbf{w}, I)$ centred on a vector $\mathbf{w}$. The KL between two such Gaussians is $\text{KL}(Q\|P) = \frac{\|\mathbf{w}\|^2}{2}$, and the vote B_Q is the linear classifier $\text{sign}(\langle \mathbf{w}, x \rangle)$ (Fig. 129). Minimizing the bound then amounts to minimizing an empirical risk plus $\frac{\|\mathbf{w}\|^2}{2}$: we recover a regularized linear classifier, similar to the SVM of Part II (Langford and Shawe-Taylor, 2002; Germain et al., 2009).

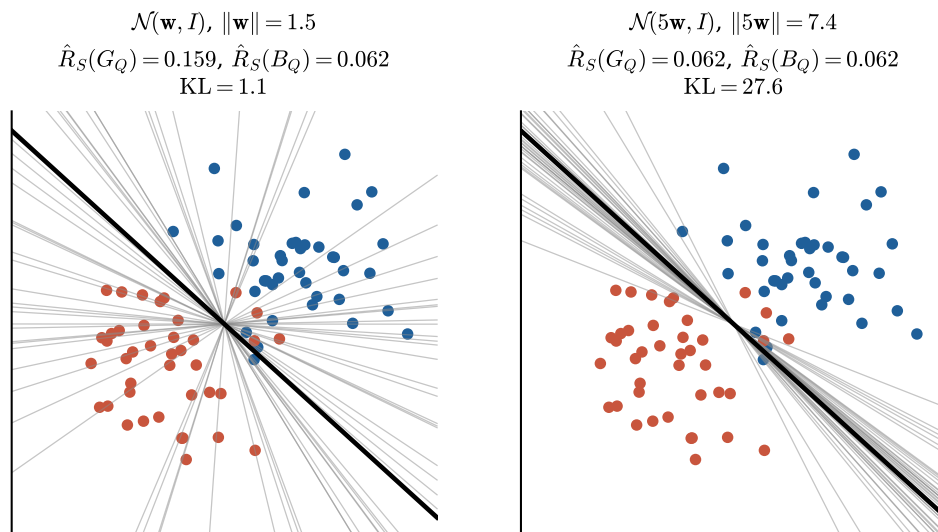


Figure 129: A Gaussian posterior $Q = \mathcal{N}(\mathbf{w}, I)$ on linear classifiers: voters drawn from Q (grey lines) and their vote B_Q (black line). Left: small $\|\mathbf{w}\|$, diverse voters, small KL. Right: large $\|\mathbf{w}\|$, voters close to the vote, larger KL.

[code: make_figures.py: fig_linear_gaussian]

Neural networks. The same Gaussian construction can be applied to the weights of a neural network. Dziugaite and Roy (2017) obtained in this way the first non-vacuous guarantees for deep networks, and Pérez-Ortiz et al. (2021) showed that minimizing such bounds gives networks that are both accurate and certified.

To go further. PAC-Bayesian theory is an active field of research: bounds for unbounded losses and dependent data, other divergences than the KL, data-dependent priors, applications to domain adaptation or to large language models (Lotfi et al., 2024). The surveys of Alquier (2024) and Guedj (2019) are good entry points.

To remember from Section 16

- Since the bounds hold for all posteriors, we can *learn* the posterior by minimizing a bound; the minimal value is a valid certificate (*self-bounding* learning).
- The KL term acts as a regularization whose weight is given by the theory.
- Everything may depend on the data except the prior and δ : the prior must be fixed before seeing the data used in the bound.
- In a random forest, each tree can be evaluated on its *out-of-bag* examples: the bounds can be computed without a test set, and every example is used for learning.
- Minimizing the first-order bound concentrates the weights and destroys the vote; minimizing the tandem bound rewards diversity and keeps a good vote.

Check your understanding

Short questions to test yourself; the answers are in Appendix A (click on the number of a question to see its answer).

- Q16.1.** Why is the value of the bound at the minimizer $\hat{Q}$ a valid certificate, although $\hat{Q}$ was chosen with the data?
- Q16.2.** May we choose the temperature λ of the Gibbs posterior by minimizing Seeger's bound on the sample? May we choose the prior in the same way?
- Q16.3.** With bootstrap samples of size m , about which fraction of the sample is out-of-bag for one tree? For two trees?
- Q16.4.** Why does minimizing the first-order bound degrade the vote of a forest?
- Q16.5.** For a Gaussian posterior $\mathcal{N}(\mathbf{w}, I)$ and the prior $\mathcal{N}(\mathbf{0}, I)$, what is the KL? Which classical regularization does it correspond to?

17 Deriving a self-bounding algorithm, step by step

About this section

Guiding question. *How do we turn a bound into a learning algorithm, concretely?*

Section 16 showed the results of bound minimization, but not how the algorithms are obtained. This section derives one completely, on our running forest. Starting from the tandem bound of *Section 15*, we follow a recipe in five steps: choose a bound, fix what must be fixed, write the quantity to minimize, solve the minimization, and certify the result. Along the way, we obtain an algorithm with a simple interpretation: each tree is penalized by the errors it shares with the other trees of the vote. The tools are the bound of Theorem 14.4, the Gibbs posterior (Proposition 14.6) and the Lagrange multipliers of Part I (Theorem 4.2).

17.1 The recipe

A self-bounding algorithm always comes from the same five steps.

1. **Choose a bound** that holds, with probability at least $1 - \delta$, for all posteriors Q at once. It will be used twice: its *shape* tells us what to minimize, its *value* certifies the result.
2. **Fix before seeing the data** everything the bound requires to be fixed: the voters (or the way they are built), the prior P , the confidence δ .
3. **Write the objective**: keep the terms of the bound that depend on Q , drop the others.
4. **Minimize the objective** over Q : with a formula when there is one, with an iterative algorithm otherwise.
5. **Certify**: evaluate the bound at the output $\hat{Q}$. Since the bound holds for all posteriors, it holds for $\hat{Q}$, even if the minimization is not exact.

A first example: the Gibbs posterior. We have already followed this recipe without saying so. With the bound of Theorem 14.4 (step 1), a prior P and a parameter λ fixed in advance (step 2), the objective is $\hat{R}_S(G_Q) + \frac{\text{KL}(Q\|P)}{\lambda}$ (step 3), the terms $\frac{\ln(1/\delta)}{\lambda}$ and $\frac{\lambda}{8m}$ not depending on Q . Proposition 14.6 solves it with a formula, the Gibbs posterior $Q_\lambda(h) \propto P(h)e^{-\lambda\hat{R}_S(h)}$ (step 4). Its certificate on the vote is the first-order bound (15.3) (step 5).

On our forest, with the OOB errors of the trees as empirical risks, this algorithm is not satisfactory (orange curves of Fig. 130). When λ grows, the Gibbs posterior concentrates on the trees with the smallest OOB error, the vote loses its diversity, and its test risk jumps from 0.044 to 0.150. This is the lesson of Section 16: to learn a *vote*, we must start from a bound that sees the diversity of the voters. We now apply the recipe to the tandem bound.

17.2 Steps 1 and 2: the bound and what is fixed

The data. The voters are the 100 trees $h_1, \dots, h_{100}$ of the forest, the prior P is uniform on their indices, and $\delta = 0.05$; all of this is fixed before looking at the out-of-bag examples (Section 16.2). For each pair of trees, we compute on the examples that are out-of-bag for both the *empirical joint error*

$$\hat{e}_S(h, h') = \text{proportion of these examples on which } h \text{ and } h' \text{ both err.}$$

These 100×100 numbers form a symmetric table, whose diagonal contains the OOB errors of the trees ($\hat{e}_S(h, h) = \hat{R}_S(h)$). The empirical joint error of a posterior is the average of the table with the weights $Q(h)Q(h')$:

$$\hat{e}_S(Q) = \sum_h \sum_{h'} Q(h) Q(h') \hat{e}_S(h, h').$$

In the bounds, m is the size of the smallest set of examples out-of-bag for a pair of trees; here $m = 423$.

The bound. We need a bound whose shape is easy to minimize. The kl form (15.4) contains the inverse $\overline{\text{kl}}^{-1}$, which has no formula. The linear bound of Theorem 14.4 is a simple sum, and it applies to the joint error exactly as Seeger’s bound does in (15.4).

Proposition 17.1: A linear tandem bound

Let P and $\lambda > 0$ be fixed before seeing the sample. With probability at least $1 - \delta$, for all distributions Q on the voters,

$$R_{\mathcal{D}}(B_Q) \leq 4 \left(\hat{e}_S(Q) + \frac{2 \text{KL}(Q \| P) + \ln \frac{1}{\delta}}{\lambda} + \frac{\lambda}{8m} \right). \quad (17.1)$$

Proof. The joint error $e_{\mathcal{D}}(Q)$ is the Gibbs risk of pairs of voters (h, h') drawn from the distribution Q^2 , for the loss “both err”, which takes values in $[0, 1]$ (Section 15). Theorem 14.4, applied to pairs with the prior P^2 , gives

$$\begin{aligned} e_{\mathcal{D}}(Q) &\leq \hat{e}_S(Q) + \frac{\text{KL}(Q^2 \| P^2) + \ln \frac{1}{\delta}}{\lambda} + \frac{\lambda}{8m} \\ &\downarrow \text{KL}(Q^2 \| P^2) = 2 \text{KL}(Q \| P) \text{ (Section 15)} \\ &= \hat{e}_S(Q) + \frac{2 \text{KL}(Q \| P) + \ln \frac{1}{\delta}}{\lambda} + \frac{\lambda}{8m}. \end{aligned}$$

We conclude with the tandem bound $R_{\mathcal{D}}(B_Q) \leq 4e_{\mathcal{D}}(Q)$ (Proposition 15.6). $\square$

17.3 Step 3: the objective

For a fixed λ , only two terms of (17.1) depend on Q . Dropping the factor 4 and the constant terms $\frac{\ln(1/\delta)}{\lambda}$ and $\frac{\lambda}{8m}$, which do not change the minimizer, we obtain the objective

$$F_{\lambda}(Q) = \underbrace{\hat{e}_S(Q)}_{\text{fit: rare common errors}} + \underbrace{\frac{2}{\lambda} \text{KL}(Q \| P)}_{\text{stay close to the prior}}, \quad (17.2)$$

to be minimized over the distributions Q on the 100 trees. As in [Section 16](#), the KL acts as a regularization, and λ sets the balance: a small λ keeps Q close to the uniform prior, a large λ lets the data speak.

In a nutshell The objective does not ask for good trees, it asks for trees that *rarely err together*. Two excellent trees that make the same mistakes have a large joint error; two average trees that err on different examples have a small one.

17.4 Step 4: solving the minimization

The optimality condition. For a voter h , let

$$\hat{e}_S(h, Q) = \sum_{h'} Q(h') \hat{e}_S(h, h')$$

be the average joint error of h with a voter drawn from Q : how often h errs *together with the vote*. It plays the role that the empirical risk $\hat{R}_S(h)$ plays for the Gibbs posterior.

Proposition 17.2: Optimality condition

If Q^* minimizes F_λ and gives a positive weight to every voter, then for every voter h ,

$$Q^*(h) = \frac{P(h) e^{-\lambda \hat{e}_S(h, Q^*)}}{Z}, \quad Z = \sum_{h'} P(h') e^{-\lambda \hat{e}_S(h', Q^*)}. \quad (17.3)$$

Proof. We minimize $F_\lambda(Q)$ under the single constraint $\sum_h Q(h) = 1$, so we apply Theorem 4.2 with the Lagrangian $\mathcal{L}(Q, \mu) = F_\lambda(Q) + \mu(\sum_h Q(h) - 1)$. We need the partial derivatives of the two terms of F_λ with respect to the weight $Q(h)$ of one voter h .

- In the double sum $\hat{e}_S(Q)$, $Q(h)$ appears in the row h and in the column h of the table; since the table is symmetric, both give the same contribution:

$$\frac{\partial \hat{e}_S(Q)}{\partial Q(h)} = 2 \sum_{h'} Q(h') \hat{e}_S(h, h') = 2 \hat{e}_S(h, Q).$$

- In $\text{KL}(Q\|P) = \sum_h Q(h) \ln \frac{Q(h)}{P(h)}$, the derivative of $q \ln \frac{q}{p}$ is $\ln \frac{q}{p} + 1$, so

$$\frac{\partial \text{KL}(Q\|P)}{\partial Q(h)} = \ln \frac{Q(h)}{P(h)} + 1.$$

At the minimum, the derivative of the Lagrangian with respect to each $Q(h)$ vanishes:

$$\begin{aligned} 2 \hat{e}_S(h, Q^*) + \frac{2}{\lambda} \left(\ln \frac{Q^*(h)}{P(h)} + 1 \right) + \mu &= 0 \\ \downarrow \text{multiply by } \frac{\lambda}{2} \text{ and isolate the logarithm} \\ \iff \ln \frac{Q^*(h)}{P(h)} &= -\lambda \hat{e}_S(h, Q^*) - c, \quad \text{with } c = 1 + \frac{\lambda \mu}{2} \\ \downarrow \text{exponentiate} \end{aligned}$$

$$\Longleftrightarrow Q^*(h) = P(h) e^{-\lambda \hat{e}_S(h, Q^*)} e^{-c}.$$

The constant e^{-c} does not depend on h ; since the weights sum to one, it equals $\frac{1}{Z}$. $\square$

Compare with the Gibbs posterior $Q_\lambda(h) \propto P(h) e^{-\lambda \hat{R}_S(h)}$: the structure is the same, but the penalty of a tree is no longer its own error, it is its error *shared with the current vote*. A tree that errs where the others are right costs little, even if its own error is high: this is how diversity enters the algorithm.

A fixed-point algorithm. Unlike the Gibbs posterior, (17.3) is not a formula: Q^* appears on both sides. Such an equation $Q = T(Q)$, where $T(Q)(h) \propto P(h) e^{-\lambda \hat{e}_S(h, Q)}$, is called a *fixed-point equation*, and the natural idea is to iterate: start from $Q = P$, compute $T(Q)$, and repeat. To avoid oscillations, we only move part of the way, by a fraction α (Algorithm 13). This is the same idea as the step size of gradient descent (Part I): too large a step may oscillate, a small step is slower but safer.

Algorithm 13: Learning the weights of a vote with the tandem bound

Input: joint errors $\hat{e}_S(h, h')$ of the voters, prior P , parameter $\lambda > 0$, step $\alpha \in (0, 1]$ (e.g. 0.1), tolerance τ (e.g. 10^{-9})

```

1  $Q \leftarrow P$ ;
2 repeat
3   for each voter  $h$  do  $s(h) \leftarrow \sum_{h'} Q(h') \hat{e}_S(h, h')$  ;    // error shared with
      the vote
4   ;
5   for each voter  $h$  do  $T(h) \leftarrow P(h) e^{-\lambda s(h)}$ ;
6    $T \leftarrow T / \sum_h T(h)$  ;                                     // normalize
7    $Q \leftarrow (1 - \alpha) Q + \alpha T$  ;                               // move part of the way
8 until the weights change by less than  $\tau$ ;
Output: the posterior  $Q$ 
```

Does it find the minimum? When all the voters are evaluated on the same examples $x_1, \dots, x_m$ (for instance on a separate validation sample), the empirical joint error is the average of the squares of the empirical Gibbs errors, $\hat{e}_S(Q) = \frac{1}{m} \sum_i W_Q(x_i)^2$ (as in the proof of Proposition 15.6). A square of a linear function of Q is convex, the KL is convex, so F_λ is convex, and a point that satisfies the optimality condition is the global minimum (Part I). With the OOB examples, the pairs are evaluated on different examples and convexity is not guaranteed, but the iterations converge quickly in practice (about 130 iterations on our forest for $\lambda \leq 1024$). In any case, step 5 does not need an exact minimum.

Pitfall The algorithm needs the 100×100 table of joint errors, not only the 100 OOB errors of the trees: these pairwise statistics are what makes it see the diversity. Using the OOB errors alone gives back the Gibbs posterior, which concentrates on a few similar trees.

17.5 Step 5: certify, and choose λ

For each λ , [Algorithm 13](#) outputs a posterior Q_λ . Which value of the bound do we report, and which λ do we keep? We use the bound in its tightest form, the kl tandem bound (15.4). It holds with probability $1 - \delta$ for *all* posteriors simultaneously, so it holds for every Q_λ , and we may compute it for each λ of a grid and keep the smallest value, at no extra cost.

The linear bound (17.1) has therefore only been used to tell us *what to minimize*; the kl bound tells us *how good the result is*. We could also certify with the linear bound itself, but λ must then be fixed in advance, or chosen in a grid of k values fixed in advance at the price of a union bound (Lemma 13.6): $\ln \frac{1}{\delta}$ becomes $\ln \frac{k}{\delta}$. On our forest, this certificate is looser (0.50 at best, for a grid of 11 values).

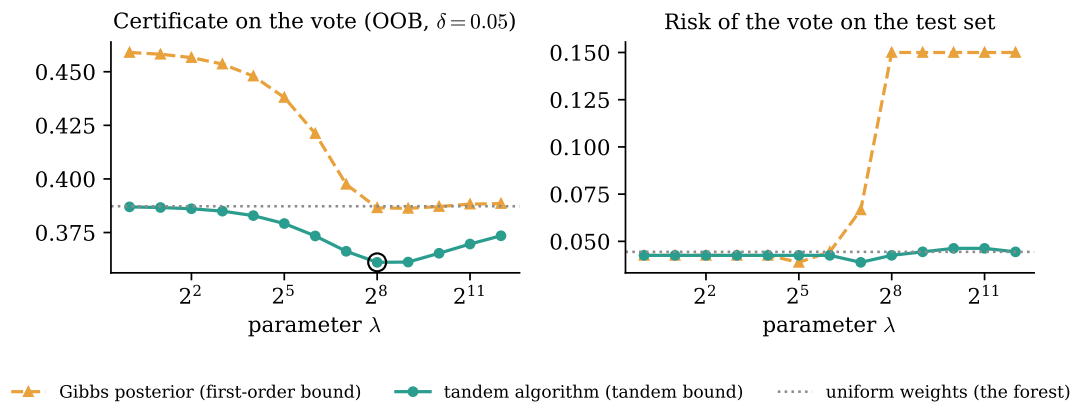


Figure 130: The two self-bounding algorithms of this section on the forest of 100 trees on **digits**, for $\lambda = 1, 2, 4, \dots, 4096$. Left: kl certificate on the risk of the vote, computed on the OOB examples (first-order bound (15.3) for the Gibbs posterior, tandem bound (15.4) for [Algorithm 13](#)); the circle marks the λ kept. Right: risk of the vote on the 540 test digits, which the algorithms never see. The dotted lines correspond to the uniform weights of the forest. [code: `make_figures.py: fig_tandem_algorithm`]

Running example We run [Algorithm 13](#) on our forest for $\lambda = 1, 2, 4, \dots, 4096$ ([Fig. 130](#) and the table below).

λ	$\hat{e}_S(Q_\lambda)$	effective trees	certificate	vote (test)
uniform	0.049	100	0.387	0.044
64	0.046	92	0.373	0.043
128	0.044	78	0.366	0.039
256	0.042	57	0.361	0.043
1024	0.040	31	0.365	0.046
4096	0.039	24	0.373	0.044

For small λ , the posterior stays close to the uniform weights. When λ grows, the joint error decreases and the weights concentrate, but slowly: even for $\lambda = 4096$, 24 trees keep a significant weight, while the Gibbs posterior keeps a single one. The certificate is minimal for $\lambda = 256$: the learned vote is certified at 0.361 (instead of 0.387 for the uniform weights), with 57 effective trees and a test risk of 0.043. We recover the tandem posterior of [Table 8](#), now with an algorithm that we have derived ourselves.

17.6 What to remember from the derivation

The same five steps apply to other bounds and other models. Two points are worth keeping in mind.

- **The bound shapes the algorithm.** The first-order bound only sees the individual errors and gives an algorithm that selects the best voters; the tandem bound sees the pairs and gives an algorithm that rewards diversity. Choosing a bound is choosing an inductive bias.
- **Optimization and certification are separate.** The algorithm may minimize a convenient version of the bound (here, the linear one), approximately; the certificate is then computed with the tightest bound available, which holds for all posteriors.

To remember from Section 17

- A self-bounding algorithm is derived in five steps: choose a bound valid for all Q , fix the prior and δ , keep the terms that depend on Q , minimize, certify.
- From the linear bound, the first-order objective $\hat{R}_S(G_Q) + \frac{\text{KL}(Q\|P)}{\lambda}$ gives the Gibbs posterior; the tandem objective $\hat{e}_S(Q) + \frac{2\text{KL}(Q\|P)}{\lambda}$ gives the condition $Q(h) \propto P(h)e^{-\lambda\hat{e}_S(h,Q)}$.
- In the tandem algorithm, a voter is penalized by the errors it shares with the vote, not by its own errors: diversity is rewarded.
- The condition is a fixed-point equation, solved by damped iterations ([Algorithm 13](#)).
- Since the bounds hold for all posteriors, the parameter λ can be chosen by minimizing the certificate, and the certificate does not require an exact minimization.
- On our forest, the derived algorithm certifies the vote at 0.361 instead of 0.387, without degrading its test risk.

Check your understanding

Short questions to test yourself; the answers are in [Appendix A](#) (click on the number of a question to see its answer).

- Q17.1.** Which terms of the linear tandem bound (17.1) are dropped to obtain the objective F_λ , and why is this allowed?
- Q17.2.** In the optimality condition, why does the derivative of $\hat{e}_S(Q)$ with respect to $Q(h)$ contain a factor 2?
- Q17.3.** What is the difference between the penalty of a tree in the Gibbs posterior and in the tandem algorithm?
- Q17.4.** Why can we not simply compute Q^* with a formula, as for the Gibbs posterior?
- Q17.5.** Is the certificate still valid if the iterations are stopped before convergence?

Q17.6. May we choose λ by minimizing the kl certificate over a grid? And by minimizing the linear certificate (17.1)?

A Answers to the “Check your understanding” questions

The number of each answer is a link back to its question.

Section 1 Probability Essentials and Concentration Inequalities

Q1.1. False. The equality only says that X and Y are uncorrelated. For X uniform on $\{-1, 0, 1\}$ and $Y = X^2$, $\mathbb{E}[XY] = \mathbb{E}[X^3] = 0 = \mathbb{E}[X] \mathbb{E}[Y]$, but Y is a function of X : $\mathbb{P}(X = 0, Y = 1) = 0 \neq \mathbb{P}(X = 0) \mathbb{P}(Y = 1) = \frac{2}{9}$.

Q1.2. The number of errors is $\text{Bin}(400, 0.2)$, so the test error $\hat{p}$ has expectation $p = 0.2$ and variance $p(1-p)/m = 0.16/400 = 4 \times 10^{-4}$, i.e. a standard deviation of 0.02: two percentage points.

Q1.3. A pmf value is the probability $\mathbb{P}(X = x) \leq 1$. A density value is not a probability: only its integrals over intervals are. A density only has to be nonnegative with a total integral equal to 1, so it can be large on a short interval, e.g. $f = 10$ on $[0, 0.1]$ for $\mathcal{U}([0, 0.1])$.

Q1.4. The square root is concave, so $\mathbb{E}[\sqrt{Z}] \leq \sqrt{\mathbb{E}[Z]}$; the exponential is convex, so $e^{\mathbb{E}[X]} \leq \mathbb{E}[e^X]$. Both inequalities are equalities when the variable is almost surely constant.

Q1.5. The training error is not: h_S depends on S , and the training error is typically too optimistic, $\mathbb{E}[\mathcal{R}_S(h_S)] < \mathbb{E}[\mathcal{R}(h_S)]$. The error on a test set T independent of S is unbiased: conditionally on S , h_S is fixed and the test losses are i.i.d. with mean $\mathcal{R}(h_S)$, so $\mathbb{E}[\mathcal{R}_T(h_S) \mid S] = \mathcal{R}(h_S)$.

Q1.6. The law of large numbers says that $\bar{X}_m \rightarrow \mu$ (almost surely), without any speed. The central limit theorem describes the size and the shape of the fluctuations: $\bar{X}_m - \mu$ is approximately $\mathcal{N}(0, \sigma^2/m)$, so it is of order $\sigma/\sqrt{m}$. Both are asymptotic.

Q1.7. The number of errors is $\text{Bin}(100, 0.01)$, which is very skewed: it equals 0 with probability $0.99^{100} \approx 0.37$, and the mean $mp = 1$ is too small for the Gaussian approximation to be accurate. In particular, when no error is observed, the estimated variance is 0 and the CLT interval reduces to $\{0\}$, which does not contain p .

Q1.8. False. The “ $m \geq 30$ ” rule of thumb is not a theorem: the central limit theorem only gives a limit, and the quality of the approximation for a given m depends on the distribution (it is poor for skewed distributions, see Fig. 4). The probability can be noticeably larger than 0.05.

Q1.9. Markov’s inequality applied to X only uses the mean and gives a $1/t$ decay. Applied to the increasing function $e^{\lambda X}$, it uses all the moments of X through the MGF, and the free parameter λ can be optimized for each t ; for bounded or Gaussian variables this gives a decay in e^{-ct^2} .

Q1.10. $\sqrt{\ln(2/0.05)/(2 \times 500)} = \sqrt{3.689/1000} \approx 0.061$, i.e. about 6 percentage points.

Q1.11. The sample size $\ln(2/\delta)/(2\varepsilon^2)$ is multiplied by 4 when ε is halved. Replacing δ by $\delta/10$ only adds $\ln 10/(2\varepsilon^2) \approx 1.15/\varepsilon^2$ examples, e.g. from 738 to 1198 for $\varepsilon = 0.05$ and $\delta = 0.05$.

Q1.12. False. They must be independent and bounded, $X_i \in [a_i, b_i]$, but they can have different distributions and different means; the inequality controls $S_m - \mathbb{E}[S_m]$, and the ranges enter through $\sum_i (b_i - a_i)^2$.

Q1.13. Hoeffding’s inequality is valid for a hypothesis fixed before seeing S . h_S is chosen because its empirical risk is small, which favours the hypotheses whose empirical risk happens to underestimate their true risk. A bound valid simultaneously for all $h \in \mathcal{H}$ (with probability $1 - \delta$) holds in particular for h_S , whatever the way it was chosen.

Q1.14. The half-width $\sqrt{(\ln K + \ln(2/\delta))/(2m)}$ goes from $\sqrt{(6.91 + 3.69)/20000} \approx 0.023$ to $\sqrt{(13.82 + 3.69)/20000} \approx 0.030$: multiplying the number of hypotheses by 1000 only increases the width by about 30%, because the dependence is in $\sqrt{\ln K}$.

Q1.15. When the variance σ^2 is much smaller than the squared range, $(b-a)^2/4$. For the 0-1 loss of a classifier with a small error rate p , $\sigma^2 = p(1-p) \approx p$: for $p = 0.01$, $m = 1000$, $\delta = 0.05$, the one-sided deviation is about 0.0097 with Bernstein against 0.0387 with Hoeffding.

Q1.16. Changing one coordinate changes the maximum by at most 1, so $c_i = 1$ and McDiarmid gives $\mathbb{P}(f - \mathbb{E}f \geq \varepsilon) \leq e^{-2\varepsilon^2/m}$, which is useless for $\varepsilon \leq 1$. McDiarmid is efficient only when every single variable has a small influence, like $1/m$ for averages.

Section 2 Linear Algebra and Convexity

Q2.1. No. It is not absolutely homogeneous, since $\|\lambda \mathbf{x}\|_2^2 = \lambda^2 \|\mathbf{x}\|_2^2 \neq |\lambda| \|\mathbf{x}\|_2^2$ when $|\lambda| \neq 0, 1$ and $\mathbf{x} \neq \mathbf{0}$, and it does not satisfy the triangle inequality: with $\mathbf{x} = \mathbf{y} = \mathbf{e}_1$, $\|\mathbf{x} + \mathbf{y}\|_2^2 = 4 > 2 = \|\mathbf{x}\|_2^2 + \|\mathbf{y}\|_2^2$. It is, however, a convex function.

Q2.2. Onto $(1, 0)^T$: $\frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{y}\|_2^2} \mathbf{y} = 2(1, 0)^T = (2, 0)^T$. Onto $(1, 1)^T$: $\frac{2+3}{2}(1, 1)^T = \frac{5}{2}(1, 1)^T = (2.5, 2.5)^T$; indeed $\mathbf{x} - \mathbf{p} = (-0.5, 0.5)^T$ is orthogonal to $(1, 1)^T$.

Q2.3. False: the inequality goes the other way. For $\mathbf{x} = (1, 1)^T$, $\|\mathbf{x}\|_1 = 2 > \sqrt{2} = \|\mathbf{x}\|_2$. The correct chain is $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{d} \|\mathbf{x}\|_2$: the L_1 unit ball is the smallest one.

Q2.4. $\|\mathbf{I}_d\|_F = \sqrt{\text{Tr}(\mathbf{I}_d)} = \sqrt{d}$. For $\mathbf{xy}^T$, whose entries are $x_i y_j$, $\|\mathbf{xy}^T\|_F^2 = \sum_{i,j} x_i^2 y_j^2 = \|\mathbf{x}\|_2^2 \|\mathbf{y}\|_2^2$, so $\|\mathbf{xy}^T\|_F = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2$.

Q2.5. $f(\mathbf{x}) = \sum_j x_j^2$, so $\partial f / \partial x_i = 2x_i$, i.e. $\nabla f(\mathbf{x}) = 2\mathbf{x}$, and $\nabla^2 f(\mathbf{x}) = 2\mathbf{I}_d$ for every $\mathbf{x}$ (the second derivatives are 2 on the diagonal and 0 elsewhere).

Q2.6. $f(1, 2) = 2$ and $\nabla f(\mathbf{x}) = (2x_1 x_2, x_1^2)^T = (4, 1)^T$ at $(1, 2)$, so $f(1.01, 2) \approx 2 + \langle (4, 1)^T, (0.01, 0)^T \rangle = 2.04$. The exact value is $1.0201 \times 2 = 2.0402$: the error, 0.0002, is of order h^2 with $h = 0.01$.

Q2.7. The direction of steepest descent is $-\nabla f(1, 1) / \|\nabla f(1, 1)\|_2$ with $\nabla f(1, 1) = (2, 6)^T$, i.e. $-(1, 3)^T / \sqrt{10}$. It is *not* the direction $-(1, 1)^T / \sqrt{2}$ of the minimizer: the gradient is orthogonal to the elliptic level set through $(1, 1)$, not aimed at its centre. This is why gradient descent zigzags on elongated level sets (Section 3).

Q2.8. The union is not convex in general: $[0, 1] \cup [2, 3]$ contains 1 and 2 but not 1.5. The intersection always is: if $\mathbf{u}, \mathbf{v} \in C_1 \cap C_2$, the segment joining them lies in C_1 and in C_2 , by convexity of each set.

Q2.9. No: it contains $(1, 1)$ and $(-1, -1)$ but not their midpoint $(0, 0)$. Its part $\{x_1 > 0, x_1 x_2 \geq 1\}$ is convex, however: it is the epigraph of the convex function $x \mapsto 1/x$ on $]0, +\infty[$ (see Proposition 2.4).

Q2.10. The unit square $[0, 1]^2$: $(\frac{1}{2}, \frac{1}{2})$ lies inside and is not a vertex. Its writing is not unique: $(\frac{1}{2}, \frac{1}{2}) = \frac{1}{2}(0, 0) + \frac{1}{2}(1, 1) = \frac{1}{2}(1, 0) + \frac{1}{2}(0, 1) = \frac{1}{4}((0, 0) + (1, 0) + (0, 1) + (1, 1))$.

Q2.11. Yes, f is convex: it is a norm on $\mathbb{R}$, or the maximum of the two linear functions x and $-x$ (Proposition 2.5). But f is not differentiable at 0, so neither Proposition 2.6 nor Proposition 2.7 applies on $\mathbb{R}$: use the definition or Proposition 2.5.

Q2.12. It is a global minimizer: by Proposition 2.6, $f(\mathbf{v}) \geq f(\mathbf{u}^*) + \langle \mathbf{0}, \mathbf{v} - \mathbf{u}^* \rangle = f(\mathbf{u}^*)$ for all $\mathbf{v}$. It need not be unique: a constant function, or the quadratic form of the middle panel of Figure 17, has a whole set of minimizers. Uniqueness holds if f is strictly convex.

Q2.13. Yes, although it has negative entries: its eigenvalues are 1 (eigenvector $(1, 1)^T$) and 3 (eigenvector $(1, -1)^T$), both positive. Equivalently, $\mathbf{z}^T \mathbf{A} \mathbf{z} = 2z_1^2 - 2z_1 z_2 + 2z_2^2 = z_1^2 + z_2^2 + (z_1 - z_2)^2 > 0$ for $\mathbf{z} \neq \mathbf{0}$.

Q2.14. $f''(x) = e^x > 0$, so f is (strictly) convex. Since $f'(x) = e^x - 1$ vanishes only at $x = 0$, the first-order condition shows that 0 is the unique global minimizer, with $f(0) = 1$. In particular $e^x \geq 1 + x$ for all x : the exponential lies above its tangent at 0.

Q2.15. Then f is not convex. If $\nabla^2 f(\mathbf{u})\mathbf{z} = \mu\mathbf{z}$ with $\mu < 0$ and $\mathbf{z} \neq \mathbf{0}$, the restriction $g(t) = f(\mathbf{u} + t\mathbf{z})$ satisfies $g''(0) = \mathbf{z}^T \nabla^2 f(\mathbf{u})\mathbf{z} = \mu \|\mathbf{z}\|_2^2 < 0$, so g is not convex near 0, and neither is f (Proposition 2.3). One negative eigenvalue at one point is enough.

Q2.16. True. If $f(u) < f(v)$ for some u, v , write v as a convex combination of u and of $w_t = u + t(v - u)$, $t > 1$: $v = (1 - \frac{1}{t})u + \frac{1}{t}w_t$. Convexity gives $f(v) \leq (1 - \frac{1}{t})f(u) + \frac{1}{t}f(w_t)$, i.e. $f(w_t) \geq f(u) + t(f(v) - f(u)) \rightarrow +\infty$ as $t \rightarrow +\infty$, a contradiction. A nonconstant convex function on $\mathbb{R}$ thus grows at least linearly on one side.

Section 3 Unconstrained and Smooth Convex Optimization

Q3.1. No: $f(\mathbf{x}) = x_1^2 - x_2^2$ has a critical point at $\mathbf{0}$, which is a saddle point; its Hessian $\text{diag}(2, -2)$ has eigenvalues of both signs (Figure 21).

Q3.2. The Hessian is $\begin{pmatrix} 2 & -4 \\ -4 & 2 \end{pmatrix}$, with eigenvalues 6 and -2 : $\mathbf{0}$ is a saddle point (along $x_1 = x_2$, $f = -2x_1^2$ decreases; along $x_1 = -x_2$, $f = 6x_1^2$ increases).

Q3.3. That it is inconclusive when the Hessian is only semidefinite: 0 is a global minimum of x^4 , a global maximum of $-x^4$, and neither for x^3 . Higher-order terms decide.

Q3.4. False: $f(x) = x^3$ has the unique critical point 0, which is not even a local minimum, and f has no minimum at all. The statement becomes true for a convex function that has a critical point.

Q3.5. By the first-order characterization of convexity (Proposition 2.6), $f(\mathbf{v}) \geq f(\mathbf{u}) + \langle \nabla f(\mathbf{u}), \mathbf{v} - \mathbf{u} \rangle = f(\mathbf{u})$ for all $\mathbf{v}$ as soon as $\nabla f(\mathbf{u}) = \mathbf{0}$.

Q3.6. When $\mathbf{X}^T \mathbf{X}$ is invertible, i.e. when $\mathbf{X}$ has rank d (linearly independent columns). The Hessian $2\mathbf{X}^T \mathbf{X}$ is then positive definite and $\boldsymbol{\theta}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$. Otherwise there are infinitely many minimizers.

Q3.7. False: Euler's equation $\mathbf{X}^T(\mathbf{p} - \mathbf{y}) = \mathbf{0}$ is nonlinear in $\boldsymbol{\theta}$ (the probabilities p_i depend on $\boldsymbol{\theta}$ through the sigmoid), so it must be solved iteratively, e.g. by gradient descent or Newton's method.

Q3.8. By Cauchy-Schwarz, among the directions $\mathbf{d}$ of a given norm, $\langle \nabla f(\mathbf{u}), \mathbf{d} \rangle$, which drives the first-order decrease of $f(\mathbf{u} - \rho \mathbf{d})$, is maximal for $\mathbf{d}$ proportional to $\nabla f(\mathbf{u})$.

Q3.9. $L = 10$ and $\rho > 2/L = 0.2$: the component u_2 is multiplied by $1 - 10\rho = -2$ at each step, so the algorithm diverges (while u_1 is multiplied by 0.7).

Q3.10. $\rho = 2/(\alpha + L) = 2/11$: the two components are multiplied by $1 - 2/11 = 9/11$ and $1 - 20/11 = -9/11$, so the error is multiplied by $(\kappa - 1)/(\kappa + 1) = 9/11 \approx 0.82$ at each iteration.

Q3.11. The derivative of $\rho \mapsto f(\mathbf{u}_k - \rho \nabla f(\mathbf{u}_k))$ vanishes at ρ_k , and this derivative is $-\langle \nabla f(\mathbf{u}_{k+1}), \nabla f(\mathbf{u}_k) \rangle$.

Q3.12. For a strictly convex φ , the graph lies strictly above its tangent at 0, so $\varphi(\rho) \leq \varphi(0) + \rho\varphi'(0)$ never holds for $\rho > 0$: with $c = 1$ no step would be accepted. With $c < 1$, all small enough steps are accepted.

Q3.13. Newton: one, whatever the condition number. Gradient descent: in general infinitely many, with a number of iterations to reach a given accuracy roughly proportional to κ .

Q3.14. False: for $f(x) = \sqrt{1 + x^2}$, $x_{k+1} = -x_k^3$, which diverges as soon as $|x_0| > 1$. Convergence is only local; damping with a line search fixes this.

Q3.15. Gradient descent: its steps are 100 times larger (L is multiplied by 100), so a step that worked may now diverge. Newton's method is unaffected, since $(100\nabla^2 f)^{-1}(100\nabla f) = (\nabla^2 f)^{-1}\nabla f$.

Q3.16. It only uses gradients and costs $\mathcal{O}(d^2)$ per iteration, instead of computing the Hessian and solving a linear system in $\mathcal{O}(d^3)$; L-BFGS even reduces the memory to $\mathcal{O}(d)$.

Section 4 Constrained and Smooth Convex Optimization

Q4.1. The set $\{h = 0\}$ of a convex but nonaffine function is in general not convex (*e.g.* the circle $\|\mathbf{x}\|_2^2 - 1 = 0$), whereas $\{\mathbf{a}^T \mathbf{x} = b\}$ is a hyperplane, hence convex.

Q4.2. No: the constraint reads $g(\mathbf{x}) = 1 - \|\mathbf{x}\|_2 \leq 0$ with g concave, and the feasible set (outside of the open unit ball) is not convex. Every point of the unit sphere is a solution: the minimizer is not unique.

Q4.3. Stationarity gives $\mathbf{x} = -\mu(1, 1)$, and the constraint gives $\mu^* = -1$, so $\mathbf{x}^* = (1, 1)$, at distance $\sqrt{2} = |b|/\|\mathbf{a}\|_2$ from the origin, as in the minimum-norm example.

Q4.4. False in general: $\nabla f(\mathbf{x}^*) = -\sum_j \mu_j^* \nabla h_j(\mathbf{x}^*)$ is only orthogonal to the feasible set. In the previous question, $\nabla f(\mathbf{x}^*) = (1, 1) \neq \mathbf{0}$.

Q4.5. The dual function is a pointwise infimum of affine functions of $(\boldsymbol{\lambda}, \boldsymbol{\mu})$, hence concave, and the constraint $\boldsymbol{\lambda} \geq \mathbf{0}$ is convex: we maximize a concave function on a convex set.

Q4.6. By weak duality, $3.0 \leq d^* \leq p^* \leq 3.2$: the point $\mathbf{x}$ is at most 0.2 above the optimal value. The dual pair is a certificate of accuracy.

Q4.7. False: a constraint qualification is needed. For $\min x$ s.t. $x^2 \leq 0$, the solution $x^* = 0$ admits no multiplier ($1 + 2\lambda \times 0 = 0$ is impossible); Slater's condition fails.

Q4.8. By complementary slackness, if $\lambda_i^* > 0$ the constraint is active, $g_i(\mathbf{x}^*) = 0$ (in the SVM, the example is a support vector). If $g_i(\mathbf{x}^*) < 0$, then $\lambda_i^* = 0$ and removing the constraint does not change the solution.

Q4.9. Keeping only the largest component, $2 - \tau = 1$ gives $\tau = 1$, and $0.5 \leq \tau$, $0 \leq \tau$: the projection is $(1, 0, 0)$, a sparse vector.

Q4.10. $\mathbf{z}/\|\mathbf{z}\|_2 = (0.6, 0.8)$ for the ball; $(1, 1)$ for the box. The two projections differ, although both sets contain the same point $(0.6, 0.8)$.

Q4.11. Clipping each coordinate gives $(1, 0)$.

Q4.12. Because the gradient does not vanish at a solution on the boundary of $\mathcal{C}$ (Figure 30); we monitor $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$ instead, which vanishes exactly at the fixed points.

Q4.13. False in general: the iterates of gradient descent may temporarily leave $\mathcal{C}$ (*e.g.* with a large step that overshoots), and the projection then modifies them. Both sequences have the same limit, though, and coincide as long as the gradient steps stay in $\mathcal{C}$.

Q4.14. It is equivalent to $\langle \rho \nabla f(\mathbf{x}^*), \mathbf{x} - \mathbf{x}^* \rangle \geq 0$ for all $\mathbf{x} \in \mathcal{C}$, and dividing by $\rho > 0$ gives the optimality condition (4.2), which does not depend on ρ .

Q4.15. One gradient evaluation plus a projection by sorting, in $\mathcal{O}(n \ln n)$ operations; the projection is cheap, so the method is well suited to learning the weights of a vote.

Section 5 Introduction

Q5.1. T : playing chess (choosing a move in each position); P : proportion of games won against a given set of opponents (or an Elo rating); E : games already played, *e.g.* against itself, as in Samuel's checkers program.

Q5.2. False. Mitchell's definition requires the performance to *improve with experience*; a program with fixed hand-written rules does not change when more data become available, however good it is.

Q5.3. Not necessarily: more *good* data generally helps, but noisy or unrepresentative data may not, and training "too much" on the same data can even degrade the performance on new data (over-fitting, Section 6).

Q5.4. Because “improving” only makes sense with respect to a criterion: a spam filter may improve its proportion of correctly sorted emails while sending more legitimate emails to the spam folder. Changing P may change which program is considered to learn best.

Q5.5. T : predict whether a user will like a movie (or rank the movies for this user); P : proportion of correct predictions, or proportion of recommended movies that are actually watched; E : the past ratings and viewing histories of the users.

Q5.6. The first one is a regression task, since the output is a real number ($y \in \mathbb{R}$); the second one is a binary classification task ($y \in \{\text{rain, no rain}\}$, encoded for instance as $\{-1, +1\}$).

Q5.7. $d = 3 \times 64 \times 64 = 12288$: one value per pixel and per color channel (R, G and B).

Q5.8. Unsupervised learning, more precisely clustering: no label (no “correct group”) is available, and the groups must be discovered from the distribution of the data only.

Q5.9. Because this distribution is unknown: we only have access to a finite sample drawn from it. The error on the distribution can only be estimated (on data that were not used for training) or bounded, which is the subject of Section 6.

Q5.10. False. We observe a finite sample $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ drawn from the distribution, not the distribution itself; supervision only means that each observed input comes with its output.

Q5.11. The labels are used only during the training phase, by the learning algorithm, to select the model h . In the prediction phase, the model only needs the feature vector $\mathbf{x}$ of the new input: its label is precisely what we want to predict.

Q5.12. They change with the lighting conditions (brightness), with the viewing angle and with the distance to the camera (apparent size), although the object is the same; the same cat can thus be represented by very different vectors. Good features should be invariant to such nuisance factors.

Q5.13. No. The positive class is, by convention, the class of interest, the one we want to detect (spam, fraud, disease), even if it is an undesirable event.

Q5.14. False. PCA only uses the inputs $\mathbf{x}_i$ (it diagonalizes their covariance matrix) and never uses a label: it is an unsupervised dimension reduction method.

Q5.15. A distribution shift: the habits of the fraudsters (and of the customers) have changed, so that the new data no longer follow the distribution of the training data. The model should be monitored and retrained on recent data.

Q5.16. In reinforcement learning there is no fixed dataset of correct answers: the data are generated by the interaction of the agent with its environment, and the agent only receives a reward (or a punishment), which says how good an action was, not which action was the correct one.

Section 6 Statistical Learning Theory

Q6.1. The 0-1 loss is piecewise constant, hence neither convex nor differentiable, and minimizing the empirical 0-1 risk is NP-hard in general. A convex surrogate leads to a convex problem that can be solved efficiently; the hinge loss $\max(0, 1 - y h(\mathbf{x}))$ is moreover an upper bound on the 0-1 loss.

Q6.2. The errors are 0, 0.5 and -0.5 . Squared loss: $(0 + 0.25 + 0.25)/3 = 1/6 \approx 0.167$. Absolute loss: $(0 + 0.5 + 0.5)/3 = 1/3 \approx 0.333$.

Q6.3. False. Unbiasedness holds for a hypothesis fixed *before* seeing S . The hypothesis h_S^* is chosen on S precisely because its empirical risk is small, so its training error is optimistic on average: $\mathbb{E}[\mathcal{R}_S^\ell(h_S^*)] \leq \mathbb{E}[\mathcal{R}_S^\ell(h_{\mathcal{H}})] = \mathcal{R}^\ell(h_{\mathcal{H}}) \leq \mathbb{E}[\mathcal{R}^\ell(h_S^*)]$, where $h_{\mathcal{H}}$ is the best hypothesis of $\mathcal{H}$ (Figure 42).

Q6.4. When $\mathcal{H}$ grows, the approximation error decreases (or stays the same) and the estimation error typically increases. The approximation error does not depend on m , whereas the estimation error typically decreases when m grows.

Q6.5. A polynomial of degree d has $d + 1$ coefficients; when $d + 1$ approaches the number m of points, the polynomial can (almost) interpolate the data, noise included. It then oscillates wildly between the training points, so that its error on new points becomes very large although its training error is close to 0.

Q6.6. Because it is an expectation with respect to the distribution $\mathcal{D}$, which is unknown. We use the empirical risk, *i.e.* the average loss on the sample S , which is an unbiased estimate of the true risk for a fixed hypothesis.

Q6.7. No: h_S^* is chosen on S to have a small empirical risk, so its training error is typically optimistic (Figure 42). This is why we need bounds that hold uniformly over $\mathcal{H}$, or an independent test set.

Q6.8. When $\lambda \rightarrow 0$, we recover plain empirical risk minimization over a large class, with a risk of over-fitting. When $\lambda \rightarrow \infty$, the parameters are forced towards $\mathbf{0}$ and the model becomes (almost) constant: under-fitting.

Q6.9. $\varepsilon = \sqrt{\ln(2/\delta)/(2m)} = \sqrt{\ln 40/2000} \approx \sqrt{3.69/2000} \approx 0.043$: with probability at least 95%, the true risk is within 4.3 points of the empirical risk.

Q6.10. The deviation term behaves as $1/\sqrt{m}$, so m must be multiplied by 4. Doubling $\ln |\mathcal{H}|$ means squaring $|\mathcal{H}|$, *i.e.* going from 10^3 to 10^6 hypotheses: multiplying $|\mathcal{H}|$ by 1000.

Q6.11. True: ε increases with $\ln(1/\delta)$, so it decreases when δ increases. But the guarantee then holds with a smaller probability $1 - \delta$: precision and confidence must be traded off.

Q6.12. Hoeffding's inequality for variables with range K reads $2e^{-2m\varepsilon^2/K^2}$, so the deviation term is multiplied by K : $\mathcal{R}^\ell(h) \leq \mathcal{R}_S^\ell(h) + K\sqrt{(\ln |\mathcal{H}| + \ln(2/\delta))/(2m)}$.

Q6.13. $d + 1 = 6$, the number of parameters (five weights and a bias).

Q6.14. The first part is true. The second part is false: one must show that *no* set of 4 points can be shattered. For linear classifiers in $\mathbb{R}^2$, four collinear points or the XOR configuration are not shattered, but this is not enough on its own: one must also check the other configurations (a point inside the triangle formed by the other three, or four points in convex position).

Q6.15. On the stability constant β of the algorithm (and on the bound K on the loss); for regularized convex problems, β behaves as $1/\lambda$ (it is $\sigma^2\kappa^2/\lambda$ in the example of the section).

Q6.16. Because it only concerns the hypothesis actually returned by the algorithm: the dependence between θ_S and S is controlled through stability (replacing one example changes the loss by at most β/m), which implies that the gap $\mathcal{R}^\ell(\theta_S) - \mathcal{R}_S^\ell(\theta_S)$ satisfies the bounded-differences condition of McDiarmid's inequality.

Section 7 Learning Procedure

Q7.1. The selected configuration would be the one that is best on these particular examples, partly by chance: the test score of the selected model would then over-estimate its generalization performance (see Proposition 11.1), and the test set would in fact have been used as a validation set.

Q7.2. A new model, learned on the whole training set with $\lambda_{i_{\max}}$; none of the 10 models learned during the cross-validation is used.

Q7.3. Over-fitting (high variance): the model fits the learning set much better than new data. One can regularize more, choose a simpler model (*e.g.* a shallower tree), collect more data or use an ensemble method such as bagging.

Q7.4. The means and variances are computed with the validation folds, which therefore influence the model: it is a (usually mild) data leakage. The scaler must be fitted on the learning folds only, *e.g.* by putting it in a pipeline with the model.

Q7.5. $4 \times 3 = 12$ candidate values, each requiring 5 models, *i.e.* 60 models, plus one final model learned on the whole training set with the selected values: 61 models.

Q7.6. False. Its estimate is almost unbiased, but it requires m trainings and its variance can be large, because the m learned models are almost identical and their errors strongly correlated. In practice, $k = 5$ or $k = 10$ is a better trade-off.

Q7.7. Hardly: the gap between the two curves is small and the validation curve no longer increases, so the model under-fits (high bias). One should rather use a more flexible model or better features.

Q7.8. Because it is the maximum of many noisy estimates: the selected configuration is partly the one that was lucky on these folds, so its score is optimistic (Proposition 11.1). The performance must be measured on data that played no role in the selection: a test set or the outer loop of a nested cross-validation.

Q7.9. Accuracy 0.99, Recall 0 and balanced accuracy $(0 + 1)/2 = 0.5$: the balanced accuracy reveals that the classifier is no better than a random guess.

Q7.10. An AUC of 0.5 means that the scores rank a positive example above a negative one only half of the time, as a random ranking does. With very rare positives, even a small False Positive Rate produces many more false alarms than true detections, so that the Precision is low; the AUC, which does not depend on the class proportions, hides this.

Q7.11. $MSE = (4 + 4 + 0 + 0)/4 = 2$, $RMSE = \sqrt{2} \approx 1.41$ and $MAE = (2 + 2 + 0 + 0)/4 = 1$.

Q7.12. False. $R^2 = 1 - MSE_h/MSE_{\bar{y}}$ is negative as soon as the model makes larger squared errors than the constant predictor $\bar{y}$, which can happen on new data (*e.g.* with an over-fitted model). It is only guaranteed to be non-negative on the training set of a least-squares model with an intercept.

Q7.13. Accuracy = $970/1000 = 0.97$, Recall = $40/50 = 0.8$, Precision = $40/60 \approx 0.667$ and $F_1 = 2TP/(2TP + FN + FP) = 80/110 \approx 0.727$.

Q7.14. True: as a harmonic mean, F_1 lies between $\min(\text{Precision}, \text{Recall})$ and $\max(\text{Precision}, \text{Recall})$. It is moreover smaller than their arithmetic mean and closer to the smaller of the two, so a model cannot get a good F_1 with only one of them high.

Q7.15. From $F_\beta = (1 + \beta^2)PR/(\beta^2P + R)$, with P the Precision and R the Recall: $F_\beta \rightarrow P$ when $\beta \rightarrow 0$ and $F_\beta \rightarrow R$ when $\beta \rightarrow +\infty$.

Q7.16. With $\exp$, an increasing function, the ranking of the examples is unchanged, so the ROC curve and the AUC are unchanged. With $-f$, the ranking is reversed: every correctly ranked pair becomes badly ranked, so the AUC becomes $1 - \text{AUC}$ (the curve is reflected through the point $(1/2, 1/2)$).

Section 8 Supervised Algorithms

Q8.1. It is 0 (each training point is its own nearest neighbor, unless two identical inputs have different labels), which says nothing about the test error: the training error of k -NN must not be used to choose k ; a validation set or a cross-validation must be used instead.

Q8.2. With two classes and an odd k , the majority vote can never be tied. With an even k , a 2-2 tie must be broken by an arbitrary convention (smallest label, random choice, or the label of the nearest neighbor), which makes the rule less natural.

Q8.3. The edge is $0.1^{1/5} \simeq 0.63$: the “neighborhood” must cover about 63% of the range of every feature to contain only 10% of the points, so it is no longer local. This is the curse of dimensionality, already visible in dimension 5.

Q8.4. The examples correctly classified strictly outside the margin, *i.e.* with $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) > 1$ (case i) of the KKT analysis). Since $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$, they do not contribute to $\mathbf{w}$, and removing them from the training set does not change the solution.

Q8.5. $\gamma = 2/\|\mathbf{w}\|_2 = 2/\sqrt{9+16} = 2/5 = 0.4$.

Q8.6. False. A larger C penalizes the slack variables more heavily, so the solution tolerates fewer margin violations, at the price of a larger $\|\mathbf{w}\|_2$, *i.e.* a narrower margin $2/\|\mathbf{w}\|_2$. It is a small C that favors a wide margin.

Q8.7. The kernel $K(\mathbf{x}, \mathbf{x}')$ is then close to 0 as soon as $\mathbf{x} \neq \mathbf{x}'$: the similarity becomes very local, the boundary wraps around individual training points, most points become support vectors and the model overfits (zero training error, poor test error).

Q8.8. The constrained form minimizes the loss over a ball. The L_1 ball has corners on the axes, where the level sets of the loss typically touch it first, so that some coordinates are exactly 0; the L_2 ball is smooth, so the contact point generally has all its coordinates nonzero (Figure 67).

Q8.9. $Pr(y = 1 \mid \mathbf{x}) = 1/(1 + e^{-\ln 3}) = 1/(1 + 1/3) = 3/4$, and the odds are $Pr(y = 1 \mid \mathbf{x})/Pr(y = 0 \mid \mathbf{x}) = e^{\ln 3} = 3$ (“3 to 1”).

Q8.10. With $d = 784$ features and an offset, each vector has $d+1 = 785$ coordinates: $(q-1)(d+1) = 9 \times 785 = 7065$ parameters with a reference class, and $q(d+1) = 7850$ for the softmax version (which is over-parameterized, since the probabilities are invariant to a common shift of the vectors).

Q8.11. The 0-1 loss is neither convex nor continuous in the parameters, and minimizing the empirical 0-1 risk is computationally hard (NP-hard in general, even for linear classifiers). Convex surrogates which upper bound it (hinge, logistic, exponential) can be minimized efficiently, and making them small also makes the 0-1 risk small.

Q8.12. Decision trees: a split $x_j \leq s$ only depends on the order of the values of feature j , which any increasing transformation (in particular standardization or min-max scaling) preserves; the threshold is simply transformed accordingly.

Q8.13. When $\lambda \rightarrow +\infty$, all the weights tend to 1, so $h(\mathbf{x})$ tends to the global mean $\frac{1}{m} \sum_i y_i$ (underfitting). When $\lambda \rightarrow 0$, the weight of the closest training point dominates all the others exponentially, so $h(\mathbf{x})$ tends to the output of the nearest neighbor of $\mathbf{x}$ (if it is unique): we recover the 1-NN regression rule (overfitting).

Q8.14. $G_N = 1 - 3 \times (1/3)^2 = 2/3$. Yes: $1 - \sum_j p_j^2$ is maximal under $\sum_j p_j = 1$ when all the p_j are equal, and the maximum for L classes is $1 - 1/L$.

Q8.15. The error is 0.2 before the split and $0.5 \times 0 + 0.5 \times 0.4 = 0.2$ after: the error criterion sees no improvement, although a pure node has been created. The Gini gain is $0.32 - 0.5 \times 0.48 = 0.08 > 0$: as a strictly concave function of the proportions, the Gini impurity (or the entropy) rewards splits that increase purity even when the majority classes do not change.

Q8.16. The k -NN rule and kernel regression (the whole training set), and the kernel SVM (the support vectors only). Linear and logistic regression, the linear SVM (through $\mathbf{w}$ and b) and decision trees (through their rules) only store a fixed set of parameters.

Section 9 Neural Networks

Q9.1. False. It stops at the first hyperplane that separates the training set, which depends on the initialization and on the order in which the examples are visited. Novikoff’s theorem only bounds the number of updates by R^2/γ^2 ; maximizing the margin is the job of the SVM.

Q9.2. By Theorem 9.1, at most $R^2/\gamma^2 = 4/0.01 = 400$ updates, whatever the dimension d , the number of examples m and the learning rate α .

Q9.3. A perceptron is a linear classifier, and no line separates the two classes of the **XOR** dataset (Fig. 74). The hidden layer computes a new representation $(x_1 \vee x_2, x_1 \wedge x_2)$ of the data, in which the two classes become linearly separable, so that a single output perceptron suffices.

Q9.4. $20 \times (10 + 1) + 20 \times (20 + 1) + 3 \times (20 + 1) = 220 + 420 + 63 = 703$ parameters.

Q9.5. An affine function $\mathbf{x} \mapsto \mathbf{W}\mathbf{x} + \mathbf{b}$ (Proposition 9.2): it has exactly the expressive power of a linear model, whatever its depth and width. Non-linear activations are needed for depth to be useful.

Q9.6. Both are equal, by invariance to the addition of a constant: $(e^1, e^2, e^3)/(e^1 + e^2 + e^3) \approx (0.090, 0.245, 0.665)$. Computing e^{103} directly may overflow in single precision, which is why we subtract $\max_k z_k$ first.

Q9.7. False. The universal approximation theorem is an existence result with a possibly huge number of hidden units, and it says nothing about learning or generalization. Some functions require exponentially fewer units with a deep network, and deep networks learn hierarchical representations that generalize better in practice.

Q9.8. The loss is $-\ln p_2 = -\ln 0.2 \approx 1.61$ and, by Proposition 9.4, $\nabla_{\mathbf{z}} \ell = \mathbf{p} - \mathbf{e}_2 = (0.7, -0.8, 0.1)$: the score of the true class is pushed up, the others down, proportionally to their probabilities.

Q9.9. Backpropagation costs a forward and a backward pass, about $3P = 3 \times 10^6$ operations per example. Finite differences require one forward pass per parameter, about $P^2 = 10^{12}$ operations: backpropagation is roughly 300 000 times cheaper.

Q9.10. All the units of a layer would compute the same function and receive the same gradient, so they would remain identical during the whole training (if all the weights are equal to 0, the errors $\delta^{(\ell)} = \sigma'_\ell(\mathbf{a}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)\top} \delta^{(\ell+1)})$ of the hidden layers are even exactly zero, so that only the last layer learns). Random initialization breaks this symmetry.

Q9.11. They are multiplied by $1/(1 - p) = 1.25$, so that their expectation is unchanged. At test time, all units are kept and nothing else is done.

Q9.12. The variance is $2/n_{\text{in}} = 2/512 \approx 0.0039$, so the standard deviation is $\sqrt{2/512} = 1/16 = 0.0625$.

Q9.13. $(3 \times 3 \times 32 + 1) \times 64 = 18\,496$ parameters. This number does not depend on the size of the images, thanks to weight sharing; only the size of the output feature maps does.

Q9.14. The convolution gives $n' = 32 - 5 + 1 = 28$, and the pooling divides it by 2: the output is 14×14 (per filter).

Q9.15. Its cell state is updated additively, $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$, so that $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ contains the term $\text{diag}(\mathbf{f}_t)$ instead of a product $\text{diag}(\tanh') \mathbf{W}$. When the forget gate is close to 1, the gradient flows through many time steps without shrinking, as in a residual connection.

Q9.16. For a fixed generator, the optimal discriminator is $D^*(\mathbf{x}) = p_{\text{data}}(\mathbf{x}) / (p_{\text{data}}(\mathbf{x}) + p_G(\mathbf{x}))$. If $p_G = p_{\text{data}}$, then $D^*(\mathbf{x}) = 1/2$ everywhere: the discriminator can do no better than a coin toss.

Section 10 Advanced Supervised Algorithms

Q10.1. By Proposition 10.1, $\mathbb{E}[(y - h(\mathbf{x}))^2] = 0.3^2 + 0.16 + 0.25 = 0.09 + 0.16 + 0.25 = 0.5$. The squared bias (0.09) and the variance (0.16) depend on the algorithm and can be reduced; the noise term (0.25) is the Bayes error: no predictor can remove it.

Q10.2. The hypothesis $h_{\mathcal{H}}$ is fixed and does not depend on S : $\mathcal{R}_S(h_{\mathcal{H}})$ is an average of m *i.i.d.* variables with mean $\mathcal{R}(h_{\mathcal{H}})$ (see Proposition 1.14 in Section 1). The hypothesis h is chosen using S , so the losses $\ell(h(\mathbf{x}_i), y_i)$ are no longer independent variables with mean $\mathcal{R}(h)$: we have to take a supremum over $\mathcal{H}$, which is precisely where the Rademacher complexity comes in.

Q10.3. Both the complexity term $2C/\sqrt{m}$ and the confidence term $\sqrt{\log(1/\delta)/(2m)}$ scale as $1/\sqrt{m}$: multiplying m by 4 divides the guaranteed gap by 2. Dividing it by 10 requires 100 times more data.

Q10.4. The variance: the model is very sensitive to its training sample (it over-fits), while its bias is small. Bagging, or a random forest if the model is a deep tree, is designed precisely to reduce the variance while keeping the bias almost unchanged.

Q10.5. A given example is never drawn in m draws with probability $(1 - 1/m)^m \rightarrow e^{-1} \simeq 0.37$. Hence about $1 - e^{-1} \simeq 63\%$ of the distinct examples appear in the bootstrap sample (some of them several times), and the remaining 37% are out-of-bag.

Q10.6. To decorrelate the trees. For T models with variance σ^2 and pairwise correlation ρ , the variance of their average is $\rho\sigma^2 + (1 - \rho)\sigma^2/T$: adding trees only kills the second term, and only a smaller ρ decreases the first one. Trees grown on bootstrap samples of the same dataset look alike, since they all split first on the most informative features.

Q10.7. False. Adding trees only reduces the Monte Carlo variability of the average: the test error decreases and reaches a plateau (Figure 96). Over-fitting in a forest is controlled by the depth of the trees, by p' and by m' , not by T . This is the opposite of boosting, where T must be chosen by early stopping.

Q10.8. If $\varepsilon_t = 1/2$, then $\alpha_t = \frac{1}{2} \ln 1 = 0$: h_t is no better than random guessing and is ignored in the vote (the weights of the examples do not change either, since $Z_t = 1$). If $\varepsilon_t < 1/2$, the weights of the misclassified examples are multiplied by $e^{\alpha_t}/Z_t = \sqrt{(1 - \varepsilon_t)/\varepsilon_t} / (2\sqrt{\varepsilon_t(1 - \varepsilon_t)}) = 1/(2\varepsilon_t) > 1$, *e.g.* by 5 when $\varepsilon_t = 0.1$.

Q10.9. It is not useless: $-h_t$ has a weighted error of 0.2. AdaBoost computes $\alpha_t = \frac{1}{2} \ln(0.2/0.8) = -\ln 2 < 0$, so that $\alpha_t h_t = (\ln 2)(-h_t)$: the classifier is used with its predictions reversed, and $Z_t = 2\sqrt{0.8 \times 0.2} = 0.8 < 1$ still decreases the bound. Only $\varepsilon_t = 1/2$ carries no information.

Q10.10. These predictions are over-optimistic (a deep tree or a 1-NN classifier has a zero training error). The meta-model would give a large weight to the base learners that best memorize the data, which are not the ones that generalize best. It must be trained on predictions made on examples that the base learners have not seen: a validation set or, better, the out-of-fold predictions of the Super Learner.

Q10.11. Since $-\partial\ell/\partial H = y - H$, the pseudo-residuals are the ordinary residuals $y_i - H_{t-1}(\mathbf{x}_i)$. No: unlike bagging, each round decreases the training loss, and boosting eventually fits the noise; T is chosen by early stopping on a validation set, together with the learning rate ν (Figure 101).

Q10.12. We get $1 - 0.9 = 0.1$ and $1 - 0.1 = 0.9$. The second example, which is misclassified, has a pseudo-residual 9 times larger, and drives the least-squares fit of the next weak learner. As AdaBoost, gradient boosting focuses on the badly predicted examples, but the pseudo-residuals of the logistic loss are bounded by 1, whereas the weights of AdaBoost can grow exponentially.

Q10.13. Here $g(\mathbf{x}_i) = \hat{y}_i^{(t-1)} - y_i$ and $h(\mathbf{x}_i) = 1$, so that, by (10.6), the value of the leaf is $\sum_i (y_i - \hat{y}_i^{(t-1)}) / (|I_j| + \lambda)$: $6/3 = 2$ (the mean residual) for $\lambda = 0$, and $6/4 = 1.5$ for $\lambda = 1$. The regularization shrinks the leaf values towards 0, all the more as the leaf contains few examples.

Q10.14. We have $d_{\mathbf{M}}(\mathbf{x}, \mathbf{x}') = |x_1 - x'_1|$: it is a pseudo-distance, but not a distance, since two distinct points with the same first coordinate are at distance 0 ($\mathbf{M}$ is PSD but singular). One can take $\mathbf{L} = \begin{pmatrix} 1 & 0 \end{pmatrix} \in \mathbb{R}^{1 \times 2}$: the metric projects the data onto the first axis and discards the second feature.

Q10.15. False. Standardization amounts to using the Mahalanobis distance with $\mathbf{M} = \text{diag}(1/s_1^2, \dots, 1/s_d^2)$: it solves the scale problem, but it does not use the labels, so a noisy feature receives the same importance as an informative one. A supervised method such as LMNN can shrink the irrelevant directions (down to 0 when $\mathbf{M}$ is singular).

Q10.16. Optimizing over $\mathbf{L}$ removes the PSD constraint, hence the eigenvalue decomposition (of cost $O(d^3)$) needed by the projection at each iteration, and a rectangular $\mathbf{L} \in \mathbb{R}^{r \times d}$ directly performs a dimensionality reduction. But the objective, which is convex (piecewise linear) in $\mathbf{M}$, becomes a non-convex function of $\mathbf{L}$, since $d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = \|\mathbf{L}(\mathbf{x} - \mathbf{x}')\|_2^2$ is quadratic in $\mathbf{L}$ and appears with a minus sign in the push term: we lose the guarantee of reaching a global minimum.

Section 11 Applications and Learning in Practice

Q11.1. 5 binary features (or 4 if one modality is dropped as a reference, which avoids a redundant column in a linear model with an intercept). The encoding $1, \dots, 5$ introduces an order and distances between the modalities (modality 5 would be “five times” modality 1), which are meaningless for a nominal feature.

Q11.2. False. The splits of a decision tree only depend on the order of the values of each feature, which is not changed by an increasing affine transformation. Standardization matters for distance-based methods, gradient-based methods and penalized models.

Q11.3. The mean of the target over this single example is 1: the encoded feature reveals the label of the example, so the model learns “this feature equal to 1 means fraud” on the training set, which does not generalize. Target encodings must be computed out-of-fold (and smoothed towards the global mean for rare modalities).

Q11.4. Because the fact that a value is missing can itself be informative (values are often not missing at random, *e.g.* a field left empty by a fraudster); after imputation by the mean or the median, this information would be lost.

Q11.5. The model would be evaluated on customers (or on a period) that it has already seen during training, which over-estimates its performance on new customers (or on the future). One must use a group split, which keeps all the examples of a customer in the same set, or a temporal split, in which the training set precedes the test set.

Q11.6. The inner validation folds, built from the outer training set, choose the hyper-parameters; the outer test folds, never seen during the selection, estimate the performance of the whole procedure.

Q11.7. $10^3 \times 5 = 5000$ models (plus one final refit). A random search with a fixed budget, *e.g.* 60 combinations (300 models), is usually almost as good, since it tests 60 distinct values of each hyper-parameter instead of 10; Bayesian optimization is another option.

Q11.8. False. By Proposition 11.1, the expectation of the maximum of the estimates is at least the maximum of the true performances: the best score is optimistic, and the more candidates and the noisier the scores, the larger the bias. Use a test set or a nested cross-validation.

Q11.9. The synthetic examples interpolate minority examples: generated before splitting, they put near-copies of the validation examples in the learning folds, which is a leakage. The validation folds must also keep the true class proportions.

Q11.10. By Proposition 11.2, $t^* = c_{FP}/(c_{FP} + c_{FN}) = 1/10$ in the first case and $1/5$ in the second one.

Q11.11. No. The model has seen a prior $\mathbb{P}(Y = 1) = 1/2$ instead of $1/100$, so it largely over-estimates the probability of the positive class on real data. The probabilities must be corrected (the odds are divided by 99) or the threshold must be tuned on validation data with the true proportions.

Q11.12. Both can be small: when one copy is permuted, the model still gets the information from the other one, so the performance decreases only partially. Permutation importance underestimates the importance of correlated features; one should permute groups of correlated features together.

Q11.13. By additivity, $h(\mathbf{x}) = \phi_0 + \sum_j \phi_j = 0.10 + 0.30 - 0.05 + 0.15 = 0.50$.

Q11.14. Not without a test: each accuracy has a 95% confidence interval of half-width about 2 points. Since both models are evaluated on the same examples, a paired test such as McNemar's test, which only looks at the examples on which they disagree, must be used.

Q11.15. The 10 training sets overlap a lot (any two of them share 8/9 of their examples), so the scores are not independent and the variance of their mean is under-estimated: the test rejects the null hypothesis too often. McNemar's test or the 5×2 cross-validated t -test are preferable.

Section 12 Why PAC-Bayes for ensemble methods?

Q12.1. It is the weighted average of the risks: $0.5 \times 0.1 + 0.3 \times 0.2 + 0.2 \times 0.3 = 0.05 + 0.06 + 0.06 = 0.17$.

Q12.2. $Q(h_1) = \frac{1}{1+3} = 0.25$ and $Q(h_2) = \frac{3}{4} = 0.75$.

Q12.3. Both can happen. It is much better when the voters err on different points (Example 12.1). It can be worse, but at most twice worse, as we will prove in Section 15.

Q12.4. It is true (any risk is at most 1), but it is *vacuous*: it gives no information.

Q12.5. The price in the bound measures how much the posterior has moved *because of the data*; this only makes sense if the reference point, the prior, does not itself depend on the data. Technically, the proof needs the prior to be independent of the sample (Section 14).

Section 13 From an error on a sample to a guarantee

Q13.1. By Markov's inequality, $\mathbb{P}(Z \geq 0.2) \leq 0.02/0.2 = 0.1$, whatever the distribution of Z .

Q13.2. $\hat{R}_S(h) = 0.05$ and the margin is $\sqrt{\ln 20/1200} = \sqrt{0.0025} = 0.05$: $R_{\mathcal{D}}(h) \leq 0.10$.

Q13.3. $\sqrt{\ln 20/(2m)} \leq 0.01$ iff $m \geq \ln 20/(2 \times 10^{-4}) \approx 15\,000$.

Q13.4. Because the fluctuations of a coin of small bias are small (variance $p(1-p)$), and the kl adapts to them, whereas Hoeffding's inequality pays for the largest possible variance, that of a fair coin.

Q13.5. Because it was chosen with the same data: among many classifiers, some look good by chance on this sample, and the selected one is likely to be one of them. Its empirical risk is optimistic.

Q13.6. $\ln K$. Multiplying K by 10 only adds $\ln 10 \approx 2.3$ to the numerator.

Section 14 The PAC-Bayesian bound

Q14.1. $\ln \frac{20}{5} = \ln 4 \approx 1.39$.

Q14.2. It is $+\infty$ (the term $Q(h) \ln \frac{Q(h)}{0}$ is infinite): the bound becomes vacuous. The prior must give a positive weight to every voter we may want to use.

Q14.3. Step 2: we exchange the expectation over the sample and the average over P , and we apply Markov's inequality to $\mathbb{E}_P e^{\lambda \Delta}$. This is only possible if P does not depend on S .

Q14.4. Because the only random event of the proof (Step 2) does not involve Q : once it holds, the change of measure gives the inequality for every Q .

Q14.5. $\ln \frac{2\sqrt{2000}}{0.05} = \ln 1789 \approx 7.49$; the margin is $\sqrt{7.49/4000} \approx 0.043$, so the bound is 0.243.

Q14.6. For $\lambda = 0$ it is the prior. For $\lambda \rightarrow \infty$, it puts all its weight on the voter(s) with the smallest empirical risk (empirical risk minimization).

Section 15 From the Gibbs classifier to the majority vote

Q15.1. $W_Q = 0.3 < \frac{1}{2}$: the vote is right.

Q15.2. $R_{\mathcal{D}}(B_Q) \leq 0.2$.

Q15.3. $e_{\mathcal{D}} = 0.2 - 0.3/2 = 0.05$, and the tandem bound is $4 \times 0.05 = 0.2$.

Q15.4. The voters always agree: $d_{\mathcal{D}} = 0$, and $e_{\mathcal{D}} = p$ (when one errs, both err). The tandem bound is $4p$, worse than the first-order bound $2p$.

Q15.5. Because the joint error is the Gibbs risk of pairs of voters drawn from Q^2 , and $\text{KL}(Q^2 \| P^2) = 2 \text{KL}(Q \| P)$.

Q15.6. $M_Q = 1 - 2W_Q$; the vote errs when $M_Q \leq 0$, i.e. $W_Q \geq \frac{1}{2}$.

Section 16 Learning with the bounds

Q16.1. Because the bound holds, with probability $1 - \delta$, simultaneously for all posteriors; in particular for $\hat{Q}$, whatever the way it was found.

Q16.2. Yes for λ : it only indexes posteriors, and Seeger's bound holds for all of them. No for the prior: it must be fixed before seeing the data used in the bound.

Q16.3. About $e^{-1} \approx 37\%$ for one tree, and $e^{-2} \approx 14\%$ for two trees (the examples out of both bootstrap samples). With bootstrap samples of size $m/2$: $e^{-1/2} \approx 61\%$ and $e^{-1} \approx 37\%$.

Q16.4. The first-order bound only sees the average error of the trees: its minimizer keeps the few best trees, which are similar, and the vote loses its diversity.

Q16.5. $\frac{\|\mathbf{w}\|^2}{2}$: the ℓ_2 (ridge) regularization of linear models and SVMs.

Section 17 Deriving a self-bounding algorithm, step by step

Q17.1. The factor 4 and the terms $\frac{\ln(1/\delta)}{\lambda}$ and $\frac{\lambda}{8m}$: for a fixed λ they do not depend on Q , so they do not change the minimizer.

Q17.2. Because $Q(h)$ appears both in the row and in the column h of the symmetric table of joint errors.

Q17.3. Gibbs: its own empirical error $\hat{R}_S(h)$. Tandem: its average joint error with the vote, $\hat{e}_S(h, Q)$, i.e. the errors it shares with the other voters.

Q17.4. Because Q^* appears on both sides of (17.3): it is a fixed-point equation, solved by iterations.

Q17.5. Yes: the kl tandem bound holds for all posteriors, in particular for the one returned, whatever its quality.

Q17.6. Yes for the kl certificate, which does not depend on λ and holds for all Q_λ . For the linear certificate, λ is a parameter of the bound: it must be fixed in advance, or chosen in a grid at the price of a union bound $(\ln \frac{k}{\delta})$.

Bibliography

- Albert, A. and Anderson, J. A. (1984). On the existence of maximum likelihood estimates in logistic regression models. *Biometrika*, 71(1):1–10.
- Alquier, P. (2024). User-friendly introduction to PAC-Bayes bounds. *Foundations and Trends in Machine Learning*, 17(2):174–303.
- Ambroise, C. and McLachlan, G. J. (2002). Selection bias in gene extraction on the basis of microarray gene-expression data. *Proceedings of the National Academy of Sciences*, 99(10):6562–6566.
- Arlot, S. and Celisse, A. (2010). A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79.
- Armijo, L. (1966). Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16(1):1–3.
- Bahnsen, A. C., Stojanovic, A., Aouada, D., and Ottersten, B. (2014). *Improving Credit Card Fraud Detection with Calibrated Probabilities*, pages 677–685.
- Bartlett, P. L. and Mendelson, S. (2003). Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3:463–482.
- Bascol, K., Emonet, R., Fromont, E., Habrard, A., Metzler, G., and Sebban, M. (2019). From cost-sensitive classification to tight F-measure bounds. In *In Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics (AISTATS-19)*.
- Batista, G. E., Bazzan, A. L., and Monard, M. C. (2003). Balancing training data for automated annotation of keywords: a case study. In *WOB*, pages 10–18.
- Behl, A., Jawahar, C., and Pawan Kumar, M. (2014). Optimizing average precision using weakly supervised data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1011–1018.
- Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854.
- Bellet, A., Habrard, A., and Sebban, M. (2013). A survey on metric learning for feature vectors and structured data. *arXiv preprint arXiv:1306.6709*.
- Bellet, A., Habrard, A., and Sebban, M. (2015). Metric learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 9(1):1–151.
- Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press, Princeton, NJ.
- Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.
- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

- Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305.
- Beyer, K., Goldstein, J., Ramakrishnan, R., and Shaft, U. (1999). When is “nearest neighbor” meaningful? In *Database Theory – ICDT’99*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer.
- Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1987). Occam’s razor. *Information Processing Letters*, 24(6):377–380.
- Bolton, R. J. and Hand, D. J. (2002). Statistical fraud detection: A review. *Statistical science*, pages 235–249.
- Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT ’92, pages 144–152. ACM.
- Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311.
- Boucheron, S., Lugosi, G., and Massart, P. (2013). *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press.
- Bousquet, O., Boucheron, S., and Lugosi, G. (2004). *Introduction to Statistical Learning Theory*, pages 169–207. Springer Berlin Heidelberg.
- Bousquet, O. and Elisseeff, A. (2002). Stability and generalization. *Journal of Machine Learning Research*, 2:499–526.
- Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA.
- Branco, P., Torgo, L., and Ribeiro, R. P. (2016). A survey of predictive modeling on imbalanced domains. *ACM Comput. Surv.*, 49(2):31:1–31:50.
- Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and regression trees*. The Wadsworth statistics/probability series. Wadsworth and Brooks/Cole Advanced Books and Software, Monterey, CA.
- Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. (2000). Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM.
- Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., and Shah, R. (1994). Signature verification using a “siamese” time delay neural network. In *Advances in neural information processing systems*, pages 737–744.
- Busa-Fekete, R., Szörényi, B., Dembczynski, K., and Hüllermeier, E. (2015). Online f-measure optimization. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 595–603. Curran Associates, Inc.

- Catoni, O. (2007). *PAC-Bayesian Supervised Classification: The Thermodynamics of Statistical Learning*, volume 56 of *IMS Lecture Notes – Monograph Series*. Institute of Mathematical Statistics.
- Cauchy, A.-L. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 25:536–538.
- Cawley, G. C. and Talbot, N. L. C. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11:2079–2107.
- Chapelle, O. (2007). Training a support vector machine in the primal. *Neural Computation*, 19:1155–1178.
- Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357.
- Chawla, N. V., Lazarevic, A., Hall, L. O., and Bowyer, K. W. (2003). Smoteboost: Improving prediction of the minority class in boosting. In *European conference on principles of data mining and knowledge discovery*, pages 107–119. Springer.
- Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794. ACM.
- Chernoff, H. (1952). A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734.
- Chopra, S., Hadsell, R., and LeCun, Y. (2005). Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 539–546. IEEE.
- Cohen, G., Hilario, M., Sax, H., Hugonnet, S., and Geissbuhler, A. (2006). Learning from imbalanced data in surveillance of nosocomial infection. *Artificial intelligence in medicine*, 37(1):7–18.
- Condorcet, N. d. (1785). *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Imprimerie Royale, Paris.
- Cortes, C. and Mohri, M. (2004). Auc optimization vs. error rate minimization. In *Advances in neural information processing systems*, pages 313–320.
- Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27.
- Cox, D. R. (1958). The regression analysis of binary sequences (with discussion). *Journal of the Royal Statistical Society*, 20:215–242.

- Cramer, J. (2003). The origins of logistic regression. *SSRN Electronic Journal*.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314.
- Dal Pozzolo, A., Boracchi, G., Caelen, O., Alippi, C., and Bontempi, G. (2015). Credit card fraud detection and concept-drift adaptation with delayed supervised information. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE.
- Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pages 233–240.
- Davis, J. V., Kulis, B., Jain, P., Sra, S., and Dhillon, I. S. (2007). Information-theoretic metric learning. In *Proceedings of the 24th international conference on Machine learning*, pages 209–216. ACM.
- Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30.
- Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923.
- Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.
- Donsker, M. D. and Varadhan, S. R. S. (1975). Asymptotic evaluation of certain Markov process expectations for large time, I. *Communications on Pure and Applied Mathematics*, 28(1):1–47.
- Dudani, S. A. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(4):325–327.
- Džeroski, S. and Ženko, B. (2004). Is combining classifiers with stacking better than selecting the best one? *Machine learning*, 54:255–273.
- Dziugaite, G. K. and Roy, D. M. (2017). Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*.
- Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *Ann. Statist.*, 7(1):1–26.
- Elkan, C. (2001). The foundations of cost-sensitive learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, pages 973–978, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Elkan, C. (2003). Using the triangle inequality to accelerate k-means. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML’03, pages 147–153. AAAI Press.
- Engel, J. (1988). Polytomous logistic regression. *Statistica Neerlandica*, 42(4):233–252.

- Fan, W., Stolfo, S. J., Zhang, J., and Chan, P. K. (1999). Adacost: misclassification cost-sensitive boosting. In *Icml*, pages 97–105.
- Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874.
- Fernández, A., Garcia, S., Herrera, F., and Chawla, N. V. (2018). Smote for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary. *Journal of Artificial Intelligence Research*, 61:863–905.
- Ferri, C., Hernández-Orallo, J., and Modroi, R. (2009). An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27–38.
- Fourure, D., Emonet, R., Fromont, E., Muselet, D., Trémeau, A., and Wolf, C. (2017). Residual conv-deconv grid network for semantic segmentation. In *Proceedings of the British Machine Vision Conference, 2017*.
- Frery, J., Habrard, A., Sebban, M., Caelen, O., and He-Guelton, L. (2017). Efficient top rank optimization with gradient boosting for supervised anomaly detection. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 20–35. Springer.
- Freund, Y. (1998). Self bounding learning algorithms. In *Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT)*, pages 247–258.
- Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.
- Freund, Y. and Schapire, R. E. (1999). A short introduction to boosting. In *In Proceedings of the Sixteenth IJCAI*, pages 1401–1406. Morgan Kaufmann.
- Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting. *The Annals of Statistics*, 28(2):337–407.
- Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232.
- Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378.
- Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., and Herrera, F. (2012). A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484.
- Gee, S. (2014). *Fraud and fraud detection: a data analytics approach*. John Wiley & Sons.
- Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58.
- Genton, M. G. (2002). Classes of kernels for machine learning: A statistics perspective. *Journal of Machine Learning Research*, 2:299–312.

- Germain, P., Lacasse, A., Laviolette, F., and Marchand, M. (2009). PAC-Bayesian learning of linear classifiers. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 353–360.
- Germain, P., Lacasse, A., Laviolette, F., Marchand, M., and Roy, J.-F. (2015). Risk bounds for the majority vote: From a PAC-Bayesian analysis to a learning algorithm. *Journal of Machine Learning Research*, 16:787–860.
- Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA. <http://www.deeplearningbook.org>.
- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.
- Grinsztajn, L., Oyallon, E., and Varoquaux, G. (2022). Why do tree-based models still outperform deep learning on typical tabular data? In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 35.
- Guedj, B. (2019). A primer on PAC-Bayesian learning. Technical Report 1901.05353, arXiv. Proceedings of the 2nd congress of the Société Mathématique de France.
- Han, H., Wang, W.-Y., and Mao, B.-H. (2005). Borderline-smote: a new over-sampling method in imbalanced data sets learning. In *International Conference on Intelligent Computing*, pages 878–887. Springer.
- Hart, P. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14(3):515–516.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, New York, second edition.
- He, H., Bai, Y., Garcia, E. A., and Li, S. (2008). Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 1322–1328. IEEE.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034.
- He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.
- He, Q. P. and Wang, J. (2008). Principal component based k-nearest-neighbor rule for semiconductor process fault detection. In *2008 American Control Conference*, pages 1606–1611.

- Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.
- Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30.
- Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67.
- Horn, R. A. and Johnson, C. R. (2012). *Matrix Analysis*. Cambridge University Press, Cambridge, second edition.
- Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.
- Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456.
- Jeni, L. A., Cohn, J. F., and De La Torre, F. (2013). Facing imbalanced data–recommendations for the use of performance metrics. In *Affective Computing and Intelligent Interaction (ACII), 2013 Humaine Association Conference on*, pages 245–251. IEEE.
- Jensen, J. L. W. V. (1906). Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica*, 30:175–193.
- Kaufman, S., Rosset, S., Perlich, C., and Stitelman, O. (2012). Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4):15:1–15:21.
- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30.
- Kearns, M. and Valiant, L. (1994). Cryptographic limitations on learning Boolean formulae and finite automata. *Journal of the ACM*, 41(1):67–95.
- Khan, A., Sohail, A., Zahoor, U., and Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*.
- Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations (ICLR)*. arXiv:1412.6980.
- Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1137–1143.
- Kolmogorov, A. N. (1933). *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin.

- Koltchinskii, V. and Panchenko, D. (2000). Rademacher processes and bounding the risk of function learning. In Giné, E., Mason, D. M., and Wellner, J. A., editors, *High Dimensional Probability II*, pages 443–457. Birkhäuser Boston.
- Konukoglu, E. and Ganz, M. (2014). Approximate false positive rate control in selection frequency for random forest. *CoRR*, abs/1410.2838.
- Koyejo, O. O., Natarajan, N., Ravikumar, P. K., and Dhillon, I. S. (2014). Consistent binary classification with generalized performance metrics. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems (NIPS-14)*, pages 2744–2752. Curran Associates, Inc.
- Krogh, A. and Vedelsby, J. (1995). Neural network ensembles, cross validation, and active learning. In *Advances in Neural Information Processing Systems 7*, pages 231–238. MIT Press.
- Lacasse, A., Laviolette, F., Marchand, M., Germain, P., and Usunier, N. (2006). PAC-Bayes bounds for the risk of the majority vote and the variance of the Gibbs classifier. In *Advances in Neural Information Processing Systems (NIPS)*, volume 19, pages 769–776.
- Langford, J. (2005). Tutorial on practical prediction theory for classification. *Journal of Machine Learning Research*, 6:273–306.
- Langford, J. and Shawe-Taylor, J. (2002). PAC-Bayes & margins. In *Advances in Neural Information Processing Systems (NIPS)*, volume 15, pages 439–446.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563.
- Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867.
- Ling, C. X., Sheng, V. S., and Yang, Q. (2006). Test strategies for cost-sensitive decision trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1055–1067.
- Liu, W. and Chawla, S. (2011). Class confidence weighted knn algorithms for imbalanced data sets. In Huang, J. Z., Cao, L., and Srivastava, J., editors, *Advances in Knowledge Discovery and Data Mining*, pages 345–356, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Lopez, V., Fernandez, A., Garcia, S., Palade, V., and Herrera, F. (2013). An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, 250:113 – 141.
- López, V., Fernández, A., Moreno-Torres, J. G., and Herrera, F. (2012). Analysis of preprocessing vs. cost-sensitive learning for imbalanced classification. open problems on intrinsic data characteristics. *Expert Systems with Applications*, 39(7):6585–6608.

- Lorenzen, S. S., Igel, C., and Seldin, Y. (2019). On PAC-Bayesian bounds for random forests. *Machine Learning*, 108:1503–1522.
- Lotfi, S., Finzi, M. A., Kuang, Y., Rudner, T. G. J., Goldblum, M., and Wilson, A. G. (2024). Non-vacuous generalization bounds for large language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 32801–32818.
- Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, pages 4765–4774.
- Masegosa, A. R., Lorenzen, S. S., Igel, C., and Seldin, Y. (2020). Second order PAC-Bayesian bounds for the weighted majority vote. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33.
- Maurer, A. (2004). A note on the PAC Bayesian theorem. Technical Report cs/0411099, arXiv.
- Maurer, A. and Pontil, M. (2009). Empirical Bernstein bounds and sample variance penalization. In *Proceedings of the 22nd Annual Conference on Learning Theory (COLT)*.
- McAllester, D. A. (1998). Some PAC-Bayesian theorems. In *Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT)*, pages 230–234.
- McAllester, D. A. (1999). Some PAC-Bayesian theorems. *Machine Learning*, 37:355–363.
- McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.
- McDiarmid, C. (1989). *On the method of bounded differences*, pages 148–188. London Mathematical Society Lecture Note Series. Cambridge University Press.
- Mercer, J. (1909). Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 209(441-458):415–446.
- Metz, C. E. (1978). Basic principles of roc analysis. In *Seminars in nuclear medicine*, volume 8, pages 283–298. Elsevier.
- Metzler, G., Badiche, X., Belkasmi, B., Fromont, E., Habrard, A., and Sebban, M. (2018a). Learning maximum excluding ellipsoids from imbalanced data with theoretical guarantees. In *Pattern Recognition Letters*, volume 112, pages 310–316. Elsevier BV.
- Metzler, G., Badiche, X., Belkasmi, B., Fromont, E., Habrard, A., and Sebban, M. (2018b). Tree-based cost sensitive methods for fraud detection. In *International Symposium on Intelligent Data Analysis (IDA- 2018)*, pages 213–224. Springer International Publishing.
- Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA.

- Mitchell, T. M. (1997). *Machine learning*, volume 1.
- Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of Machine Learning*. The MIT Press.
- Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2018). *Foundations of Machine Learning*. MIT Press, 2nd edition.
- Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142.
- Narasimhan, H., Ramaswamy, H., Saha, A., and Agarwal, S. (2015). Consistent multiclass algorithms for complex performance measures. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, volume 37 of *Proceedings of Machine Learning Research*, pages 2398–2407, Lille, France. PMLR.
- Nesterov, Y. (2018). *Lectures on Convex Optimization*, volume 137 of *Springer Optimization and Its Applications*. Springer, Cham, 2nd edition.
- Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, New York, 2nd edition.
- Novikoff, A. B. J. (1962). On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622. Polytechnic Institute of Brooklyn.
- Parambath, S. P., Usunier, N., and Grandvalet, Y. (2014). Optimizing f-measures by cost-sensitive classification. In *Advances in Neural Information Processing Systems (NIPS-14)*, pages 2123–2131.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.
- Pérez-Ortiz, M., Rivasplata, O., Shawe-Taylor, J., and Szepesvári, C. (2021). Tighter risk certificates for neural networks. *Journal of Machine Learning Research*, 22(227):1–40.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.
- Prokhorenkova, L., Gusev, G., Vorobey, A., Dorogush, A. V., and Gulin, A. (2018). CatBoost: Unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31.
- Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.
- Rijsbergen, C. J. V. (1979). *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition.
- Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407.

- Rokach, L. and Maimon, O. (2005). Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487.
- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.
- Safavian, S. R. and Landgrebe, D. (1991). A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674.
- Saito, T. and Rehmsmeier, M. (2015). The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PLoS ONE*, 10(3):e0118432.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.
- Sanderson, M. (1994). Word sense disambiguation and information retrieval. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '94, pages 142–151, New York, NY, USA. Springer-Verlag New York, Inc.
- Sauer, N. (1972). On the density of families of sets. *Journal of Combinatorial Theory, Series A*, 13(1):145–147.
- Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.
- Schapire, R. E. and Freund, Y. (2012). *Boosting: Foundations and Algorithms*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA.
- Schapire, R. E., Freund, Y., Bartlett, P., and Lee, W. S. (1998). Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686.
- Scholkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA.
- Seeger, M. (2002). PAC-Bayesian generalisation error bounds for Gaussian process classification. *Journal of Machine Learning Research*, 3:233–269.
- Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- Shawe-Taylor, J. and Williamson, R. C. (1997). A PAC analysis of a Bayesian estimator. In *Proceedings of the 10th Annual Conference on Computational Learning Theory (COLT)*, pages 2–9.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

- Strang, G. (2016). *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA, fifth edition.
- Thiemann, N., Igel, C., Wintenberger, O., and Seldin, Y. (2017). A strongly quasiconvex PAC-Bayesian bound. In *Proceedings of the 28th International Conference on Algorithmic Learning Theory (ALT)*, volume 76 of *PMLR*, pages 466–492.
- Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.
- Tomek, I. (1976). Two modifications of cnn. *IEEE Trans. Systems, Man and Cybernetics*, 6:769–772.
- Torgo, L. and Lopes, E. (2011). Utility-based fraud detection. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1517–1522.
- Torgo, L. and Ribeiro, R. (2007). Utility-based regression. In *Knowledge Discovery in Databases: PKDD 2007*, pages 597–604. Springer Berlin Heidelberg.
- Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.
- Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.
- Van der Laan, M. J., Polley, E. C., and Hubbard, A. E. (2007). Super learner. *Statistical applications in genetics and molecular biology*, 6(1).
- Vapnik, V. and Chervonenkis, A. (1982). Necessary and sufficient conditions for the uniform convergence of means to their expectations. *Theory of Probability & Its Applications*, 26(3):532–553.
- Vapnik, V. and Cortes, C. (1995). Support-vector networks. *Machine Learning*, 20:273–297.
- Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Heidelberg.
- Vapnik, V. N. (1998). *Statistical Learning Theory*. John Wiley & Sons, New York.
- Vapnik, V. N. and Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280.
- Varma, S. and Simon, R. (2006). Bias in error estimation when using cross-validation for model selection. *BMC Bioinformatics*, 7:91.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008.
- Viallard, P., Germain, P., Habrard, A., and Morvant, E. (2021). Self-bounding majority vote learning algorithms by the direct minimization of a tight PAC-Bayesian C-bound. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 12976 of *LNCS*, pages 167–183. Springer.

- Viola, R., Emonet, R., Habrard, A., Metzler, G., Riou, S., and Sebban, M. (2019). An adjusted nearest neighbor algorithm maximizing the f-measure from imbalanced data. In *In Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI-2019)*.
- von Luxburg, U. and Bousquet, O. (2004). Distance-based classification with lipschitz functions. *J. Mach. Learn. Res.*, 5(Jun):669–695.
- Wager, S., Wang, S., and Liang, P. (2013). Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems*, volume 26.
- Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer, New York.
- Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, 26(4):359–372.
- Weinberger, K. Q. and Saul, L. K. (2009). Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, 10:207–244.
- Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, 5(2):241–259.
- Xing, E. P., Ng, A. Y., Jordan, M. I., and Russell, S. (2002). Distance metric learning with application to clustering with side-information. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, pages 505–512. MIT Press.
- Zantedeschi, V., Emonet, R., and Sebban, M. (2016). Metric learning as convex combinations of local models with generalization guarantees. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2017). Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations (ICLR)*.
- Zhao, M.-J., Edakunni, N. U., Pocock, A. C., and Brown, G. (2013). Beyond fano’s inequality: bounds on the optimal f-score, ber, and cost-sensitive risk and their implications. *Journal of Machine Learning Research*, 14:1033–1090.